

Walter Mora F.
Otro Autor

Introducción a los Métodos Numéricos.

Implementaciones en **R**

Primera edición



Revista digital

Matemática, Educación e Internet. (<http://tecdigital.tec.ac.cr/revistamatematica/>).

Copyright© Revista digital Matemática Educación e Internet (<http://tecdigital.tec.ac.cr/revistamatematica/>).
Correo Electrónico: wmora2@itcr.ac.cr
Escuela de Matemática
Instituto Tecnológico de Costa Rica
Apdo. 159-7050, Cartago
Teléfono (506)25502225
Fax (506)25502493

Mora Flores, Walter.
Introducción a los métodos numéricos. Implementaciones en R, 1ra ed.
– Escuela de Matemática, Instituto Tecnológico de Costa Rica. 2015.
396 pp.
ISBN Obra Independiente: 978-9968-641-13-5
1. Métodos Numéricos. 2. Programación 3. Algoritmos.

Derechos reservados © 2015

Revista digital

Matemática, Educación e Internet.

<http://www.tec-digital.itcr.ac.cr/revistamatematica/>.

Photos by: Viviana Loaiza. Parque Nacional Chirripó, Costa Rica.



Límite de responsabilidad y exención de garantía: El autor o los autores han hecho su mejor esfuerzo en la preparación de este material. Esta edición se proporciona “tal cual”. Se distribuye gratuitamente con la esperanza de que sea útil, pero sin ninguna garantía expresa o implícita respecto a la exactitud o completitud del contenido. La Revista digital Matemáticas, Educación e Internet es una publicación electrónica. El material publicado en ella expresa la opinión de sus autores y no necesariamente la opinión de la revista ni la del Instituto Tecnológico de Costa Rica.

Primera edición, Febrero 2015



Índice general

Prólogo 8

1	Programación con R	11
1.1	Introducción	11
1.2	Sesiones en RStudio	12
1.3	R como calculador	15
1.4	Definir variables y asignar valores	17
1.5	Funciones I.	18
1.6	Vectores	20
1.7	Paquetes	34
1.8	Matrices y arreglos	35
1.8.1	Operaciones con matrices	42
1.9	Función apply()	44
1.10	Condicionales y Ciclos	48
1.11	La eficiencia de la vectorización.	53
1.12	Funciones II	54
1.13	Expresiones, tiras y funciones	57
1.14	Gráficos en R	63
1.15	Correr un programa (script)	69
1.16	R en la nube.	71

2	Aritmética del Computador y Errores	73
2.1	Introducción	73
2.2	Aritmética del computador.	79
2.3	Cancelación	81
2.4	Propagación del Error	85
3	Polinomio de Taylor	89
3.1	Estimación del error	90
3.1.1	Implementación	93
4	Ecuaciones no lineales.....	95
4.1	Orden de convergencia	96
4.2	Método de Punto Fijo	98
4.3	El método de Bisección	109
4.4	Algoritmo e Implementación.	112
4.5	Orden de convergencia y número de iteraciones.	116
4.6	El Método de Newton	117
4.7	Método de Newton: Algoritmo e Implementación.	123
4.8	Un método híbrido: Newton-Bisección.	128
4.9	Método de la Secante	130
4.10	El Método de la Falsa Posición	134
5	Sistema de ecuaciones Lineales	137
5.1	Eliminación Gaussiana	137
5.2	Análisis de error.	145
5.3	Descomposición LU.	148
5.4	Refinamiento iterativo.	159
5.5	Grandes sistemas y métodos iterativos.	159
5.5.1	Sistemas $m \times n$	168
6	Interpolación Polinomial	171
6.1	Introducción	171
6.2	Interpolación polinomial.	173
6.3	Forma de Lagrange del polinomio interpolante.	174

6.4	Forma modificada y forma baricéntrica de Lagrange.	181
6.5	Forma de Newton para el polinomio interpolante.	185
6.6	Diferencias Divididas de Newton.	185
6.7	Forma de Lagrange vs Forma de Newton.	192
6.8	Estimación del error.	193
6.9	Error en interpolación lineal.	194
6.10	Error en interpolación cuadrática	195
6.11	Error en interpolación cúbica	195
6.12	Error con interpolación con polinomios de grado n.	197
6.13	Otros casos.	198
6.14	(*) Interpolación Iterada de Neville	199
6.15	Trazadores Cúbicos (Cubic Splines).	203
7	Derivación e integración numérica	213
7.1	Derivación numérica	213
7.2	Integración numérica	219
7.3	Fórmulas de Newton-Cotes.	220
7.4	Regla del Trapecio.	221
7.5	Regla del Simpson.	224
7.6	Método de Romberg.	229
7.6.1	Matriz de Romberg	230
7.7	Cuadratura Gaussiana.	233
7.8	Integrales Impropias.	236
7.9	Integración con R	236
8	Ecuaciones Diferenciales Ordinarias	243
8.1	Método de Euler	244
8.2	Algoritmo e implementación	246
8.3	Métodos de Taylor de orden superior.	247
8.3.1	implementación	250
8.4	Métodos de Runge-Kutta.	251
8.4.1	Algoritmo e implementación	252
8.4.2	Paquete “deSolve” de R	254

8.5	Algunos Detalles Teóricos.	258
8.6	Estimación del error	260
	Bibliografía	261
9	Soluciones de los ejercicios	265



Prólogo

El propósito de este libro es la implementación de métodos numéricos básicos usando el lenguaje de programación (libre) **R**. El curso esta orientado a estudiantes con poco conocimiento de programación.

Aunque **R** es un lenguaje y un entorno de programación para análisis estadístico y gráfico, **R** también puede usarse como herramienta de cálculo numérico, campo en el que puede ser tan eficaz como otras herramientas específicas tales como GNU OCTAVE y su equivalente comercial, MATLAB. En este libro se expone la teoría, a veces con justificaciones teóricas, se presentan varios ejemplos y al final del cada capítulo, se dedica tiempo a los algoritmos y las implementaciones.

Cartago, Febrero 2015.

W. MORA F.....



1 — Programación con R

1.1 Introducción

R es a la vez un entorno interactivo de gran alcance para el análisis de datos, visualización y modelado y es un lenguaje diseñado y construido para dar soporte a estas tareas (desplegar datos, resúmenes, estimación de modelos, simulación, cálculo numérico, etc.) con código sencillo y natural. Es software libre y fue desarrollado por Robert Gentleman y Ross Ihaka del Departamento de Estadística de la Universidad de Auckland en 1993 (a partir del lenguaje S).

Instalación

R es software libre y se puede descargar (Mac, Linux y Windows) en <http://www.r-project.org/>. Una interfaz de usuario para R podría ser RSTUDIO y se puede descargar (Mac, Linux y Windows) en <http://www.rstudio.com/>. La instalación es directa en los tres casos.

La documentación oficial de R se puede ver en <http://stat.ethz.ch/R-manual/R-devel/library/base/html/00Index.html>. Una guía rápida bastante recomendable está en

<http://www.statmethods.net/index.html>

También hay varios manuales gratuitos de R, la manera fácil sería googlear¹ “manual de R” o algo por el estilo. Por ejemplo, un buen manual está en <http://cran.r-project.org/doc/manuals/r-release/R-intro.html>.

¹Googlear es un neologismo común entre los usuarios de internet que utilizan el buscador Google.

1.2 Sesiones en RStudio

Uno de los ambientes de desarrollo para usar con **R** es **RStudio**. Este entorno de trabajo viene dividido en varios paneles y cada panel tiene varias “pestañas”, por ejemplo

Console: Aquí es donde se pueden ejecutar comandos de **R** de manera interactiva

History: Histórico con las variables y funciones definidas, etc. (puede ser guardado, recargado, etc.)

Plots: Ventana que muestra los gráficos de la sesión

Help: Esta es una ventana de ayuda, aquí aparece la información cuando se pide (**seleccione un comando y presione F1**)

Files: Manejador de información, aquí podemos localizar, cargar, mover, renombrar, etc.

Packages: Instalar o cargar paquetes

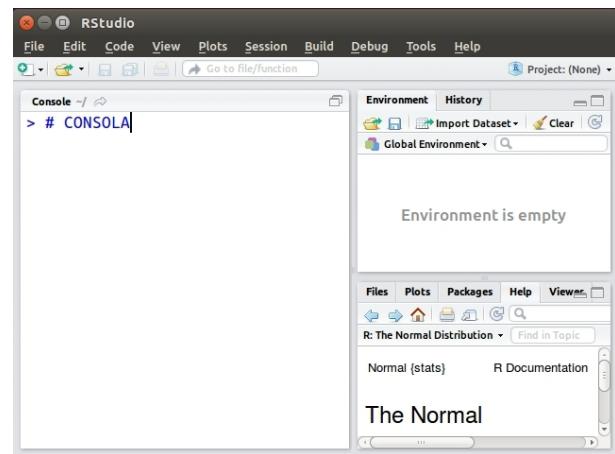


Figura 1.1: Sesión **R** inicial con RStudio

■ **Caminos cortos con las teclas:** Una lista de completa de juegos de teclas y su acción se puede ver en <https://support.rstudio.com/hc/en-us/articles/200711853>.

Por ejemplo,

- a.) **Ctrl+L:** Limpia la consola
- b.) **Ctrl+Shift+C:** Comentar código seleccionado

■ **Workspace.** El entorno de trabajo (workspace) incluye todos los objetos definidos por el usuario (vectores, matrices, data frames, listas, funciones). Al final de una sesión de **R**, el usuario puede guardar una imagen del espacio de trabajo actual que se vuelve a cargar automáticamente la próxima vez **R** se inicia

■ **Consola.** Para empezar podríamos usar la consola para digitar y ejecutar algunos comandos (presionando **Enter** para ejecutar). La consola se puede limpiar presionando **Ctrl+L**. Los primeros ejemplos de este capítulo se pueden ejecutar en la consola.

■ **Scripts.** Por defecto, **R** inicia una sesión interactiva con la entrada desde teclado y la salida a la pantalla (en el panel de la consola). Sin embargo, para programar y ejecutar código más extenso (con funciones y otro código), lo que debemos hacer es abrir un “**R script**” nuevo con el menú **File - New File - R Script**.

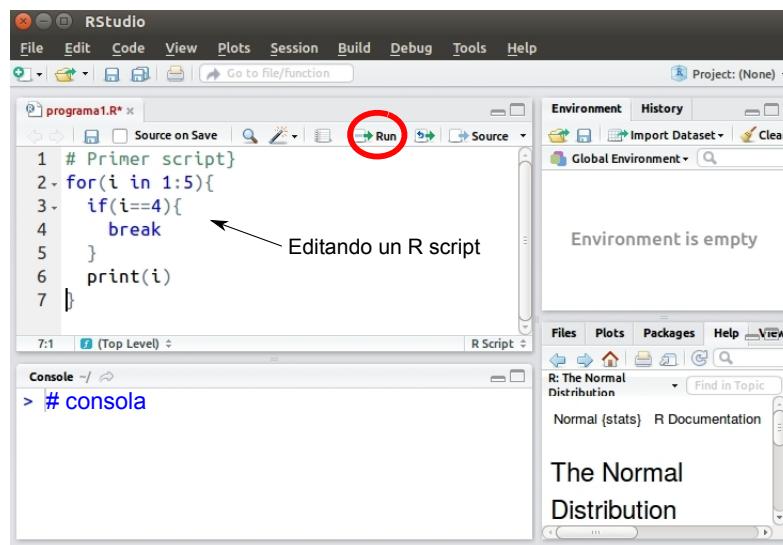


Figura 1.2: Editando un R script

Los scripts son archivos .R con código R. Posiblemente funciones definidas por nosotros (nuestros programas) y otros códigos. Para ejecutar este código (o parte de este código si los hemos seleccionado con el mouse) se presiona sobre el ícono run (para “lanzarlo a la consola”)

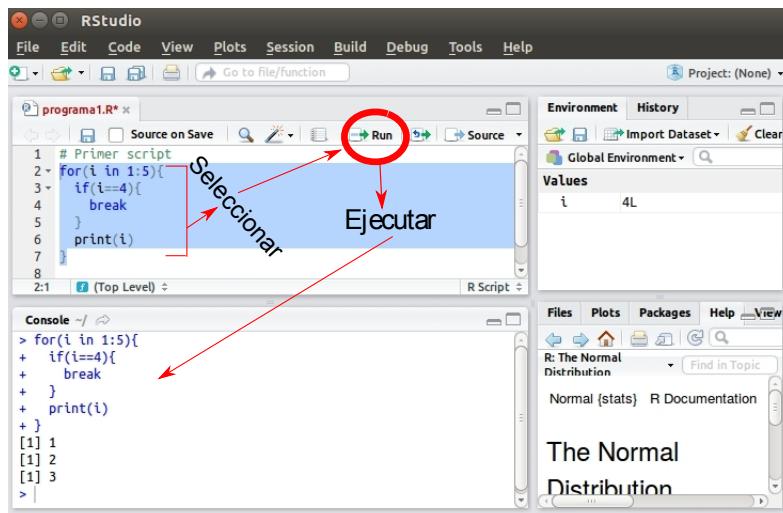


Figura 1.3: Ejecución del código seleccionado

Para ejecutar todo el código se usa **Ctrl+Shift+Enter** o presionar el botón  Source ▾

■ **Ayuda (Help)**. Para obtener ayuda de un comando o una función, se selecciona y se presiona **F1**. También un comando se puede completar con la tecla **Tab** (adicionalmente se obtiene una descripción mínima).

Generar reportes

Se puede generar un reporte de la sesión en formato **.pdf**, **.docx** o **.html** como se muestra en la figura que sigue,

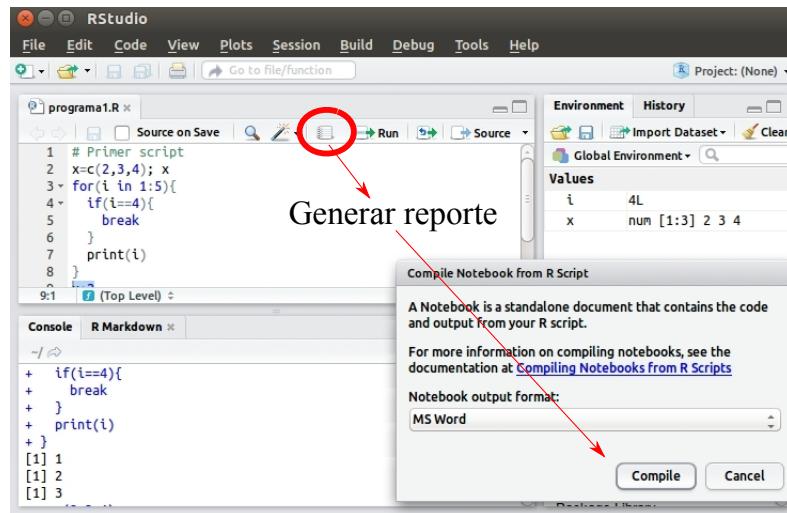


Figura 1.4: Generar un reporte

■ **Reportes profesionales.** Se puede generar un reporte profesional (y posiblemente dinámico) para una presentación, un artículo, un libro, etc. usando **LATEX** y el paquete **knitr**. Para esto puede consultar el libro Mora W. "Edición de textos científicos con *LaTeX*". ITCR. (pp. 214 - 218). El libro lo puede descargar en https://tecdigital.tec.ac.cr/revistamatematica/Libros/LaTeX_LaTeX_2013.pdf

■ **Presentaciones dinámicas.** **Shiny** es un paquete de **R** para crear aplicaciones web interactivas (apps) directamente desde **R**. Puede consultar <http://shiny.rstudio.com/tutorial/lesson1/>.

■ **Paquetes.** Mucho del poder de **R** viene de la inmensa cantidad de paquetes conteniendo código y datos para situaciones especiales. En la pestaña **Packages** se puede inspeccionar los paquetes instalados.

Los paquetes con casilla marcada (✓) indican que el paquete está cargado y se puede usar. Para buscar y/o instalar nuevos paquetes se usa la pestaña **Install**.

Una lista (con su respectiva descripción) de qué paquetes son útiles en métodos numéricos se puede obtener en la página "RAN Task View: Numerical Mathematics",

<http://cran.r-project.org/web/views/NumericalMathematics.html>

Files	Plots	Packages	Help	Viewer
<input type="checkbox"/> Foreign	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...			
<input type="checkbox"/> Formula	Extended Model Formulas	1.1-1		
<input checked="" type="checkbox"/> graphics	The R Graphics Package	3.1.0		
<input checked="" type="checkbox"/> grDevices	The R Graphics Devices and Support for Colours and Fonts	3.1.0		
<input type="checkbox"/> grid	The Grid Graphics Package	3.1.0		
<input type="checkbox"/> Hmisc	Harrell Miscellaneous	3.14-0		
<input type="checkbox"/> KernSmooth	Functions for kernel smoothing for Wand & Jones (1995)	2.23-12		
<input type="checkbox"/> lattice	Lattice Graphics	0.20-27		

Figura 1.5: Pestaña Packages

1.3 R como calculador

R se puede usar como un “calculador simbólico” sin tener que programar nada.

- Constantes: `pi` (π)
- Operadores: `<,>, <=, >=, !=, !(Not), | (OR), &(And), == (comparar)`
- Operaciones aritméticas: `+, -, *, /, \^` (potencias), `%%` (mod = resto de la división entera), y `%/%` (división entera).
- Logaritmos y exponentiales: `log` (logaritmo natural), `log(x,b)` ($\log_b x$) y `exp(x)` (e^x).
- Funciones trigonométricas: `cos(x)`, `sin(x)`, `tan(x)`, `acos(x)`, `asin(x)`, `atan(x)`, `atan2(y,x)` con x , y en radianes.
- Funciones misceláneas: `abs(x)`, `sqrt(x)`, `floor(x)`, `ceiling(x)`, `max(x)`, `sign(x)`

Ejemplo 1.1

a.) Calcular $\cos(\pi/6 + \pi/2) + e^2$

```
cos(pi/6+pi/2)+exp(1)^2 #Comentario: e=exp(1)
#[1] 6.889056
```

b.) Calcular $\cos(\pi/6 + \pi/2) + e^2 * \log_2 5 + \arccos(1/\sqrt{2})$

```
cos(pi/6+pi/2)+exp(1)^2*log(2,5)+acos(1/sqrt(2))
#[1] 3.467691
```

c.) Mensajes de error

```
acos(2/sqrt(2)) # Error pues, en acos(x), debe ser -1 <= x <= 1
#[1] NaN
#Mensajes de aviso perdidos
#In acos(2/sqrt(2)) : Se han producido NaNs
#> # Para pedir ayuda use ? antes de símbolo
#> ?NaN
```

En los ejemplos anteriores, R ve la salida como un vector y [1] indica que es la primera entrada del vector de salida.

Visualización numérica

■ **Dígitos.** R despliega los números con siete dígitos. Esto se puede cambiar con `options(digits=k)`. La opción permanece hasta que es cambiada o hasta que se reinicie R .

■ Código R 1.1:

```
1/3.0  
# [1] 0.3333333  
options(digits=3)  
1/3  
# [1] 0.333
```

■ **Redondeo.** `round(x,n)` redondea x a n decimales. Por defecto, $n = 0$.

■ Código R 1.2:

```
round(67.3)  
#[1] 67  
round(67.3787662,2) # sería 67.40  
#[1] 67.4
```

■ **Dígitos significativos.** `signif(x,n)` redondea x a n dígitos significativos. Por defecto, $n = 6$.

■ Código R 1.3:

```
options(digits=7) #retornar a siete dígitos  
signif(67.3787662,2)  
#[1] 67  
signif(67.3787662) #defecto n=6  
#[1] 67.3788
```

■ **Deshabilitar la notación científica.** A veces es adecuado no usar la notación científica para visualizar mejor los datos numéricos. Podemos deshabilitar esta notación con `options(scipen=999)` . Para habilitarla de nuevo usamos `options(scipen=0)`

■ Código R 1.4:

```
exp(-30)  
#[1] 9.357623e-14
```

```
options(scipen = 999) # Deshabilitar la notación científica
exp(-30)
#[1] 0.00000000000009357623
options(scipen = 0)
```

1.4 Definir variables y asignar valores

Para definir y/o asignar valores a una variable se usa "`=`" o también `<-`. R es sensitivo a mayúsculas y los nombres de las variables deben iniciar con letras o con `.`

Las instrucciones en una sola línea se pueden separar por "punto y coma". Se puede definir una lista de variables separadas por `;`:

■ Código R 1.5: Asignación de valores a las variables `x0` y `x1`

```
e = exp(1); x = 0.003           # asignación. Se puede usar ";" para separar instrucciones
x0 = e^(2*x)
txt = "El valor de x0 es "      # asignación string
cat(txt, x0)                   # "cat" concatena y convierte a string
# El valor de x0 es 1.006018
```

Podemos imprimir el valor de una variable usando paréntesis

■ Código R 1.6: Asignación de valores a las variables `x0` y `x1`

```
x0 = 1
(x1 = x0 - pi*x0 + 1 )
#[1] -1.141593
```

Nombrar y remover variables. Con `ls()` obtenemos las variables que hemos definido y con `remove(variable.names)` eliminamos las variables. Los objetos que creamos en la sesión, permanecen hasta que explícitamente los eliminemos. En general, antes de correr un programa debemos limpiar el espacio de trabajo para empezar siempre "desde el principio."

La lista de todos los objetos definidos actualmente se obtiene con `ls()`. Para eliminar un objeto `x`, usamos `rm(x)`. Para eliminar todos los objetos definidos en la actualidad se usa `rm(list= ls())`.

■ Código R 1.7:

```
# Objetos definidos
ls()
# [1] "e"   "txt" "x"   "x0" ...
# Eliminar todos los objetos
rm(list= ls())
```

■ **Imprimir.** Se puede usar, entre otros, el comando `print()` y `cat()` para imprimir (en la consola o a un archivo). `cat()` imprime y retorna `NULL`. `print()` imprime y se puede usar lo que retorna. Por ejemplo

■ Código R 1.8: cat

```
x0 = 1
x1 = x0 - pi*x0 + 1
  cat("x0 =", x0, "\n", "x1 =", x1)    ## "\n" = cambio de linea
# x0 = 1
# x1 = -1.141593
x2 = print(x1)                      # print imprime
# [1] -1.141593                      # y se puede usar el valor que retorna
(x2+1)
# [1] -0.1415927
```

1.5 Funciones I.

Las funciones se declaran usando la directiva `function(a1,a2,...,an){... código...}` y es almacenada como un objeto. Las funciones tienen argumentos y estos argumentos podrían tener valores por defecto. La sintaxis sería algo como

■ Código R 1.9: Funciones

```
nomrefun = function(a1,a2,...,an) {
# código ...
instrucc-1
instrucc-2
# ...
return(salida) #valor que retorna (o también la última instrucción, si ésta retorna algo)
}
```

Por ejemplo, $f(x) = \cos x$ se implementa como

```
fun = function(x){ cos(x)}
# llamada por nombre "fun"
fun(pi/3)
# [1] 0.5
```

Como el código solo tiene una instrucción, se puede obviar las llaves: `f = function(x)cos(x)`

Otro ejemplo es la función discriminante, si $P(x) = ax^2 + bx + c$ entonces el discriminante es $\Delta = b^2 - 4ac$.

■ Código R 1.10:

```
d = function(a,b,c) b^2-4*a*c
# llamar a la función por nombre
d(2,2,1)
#[1] -4
```

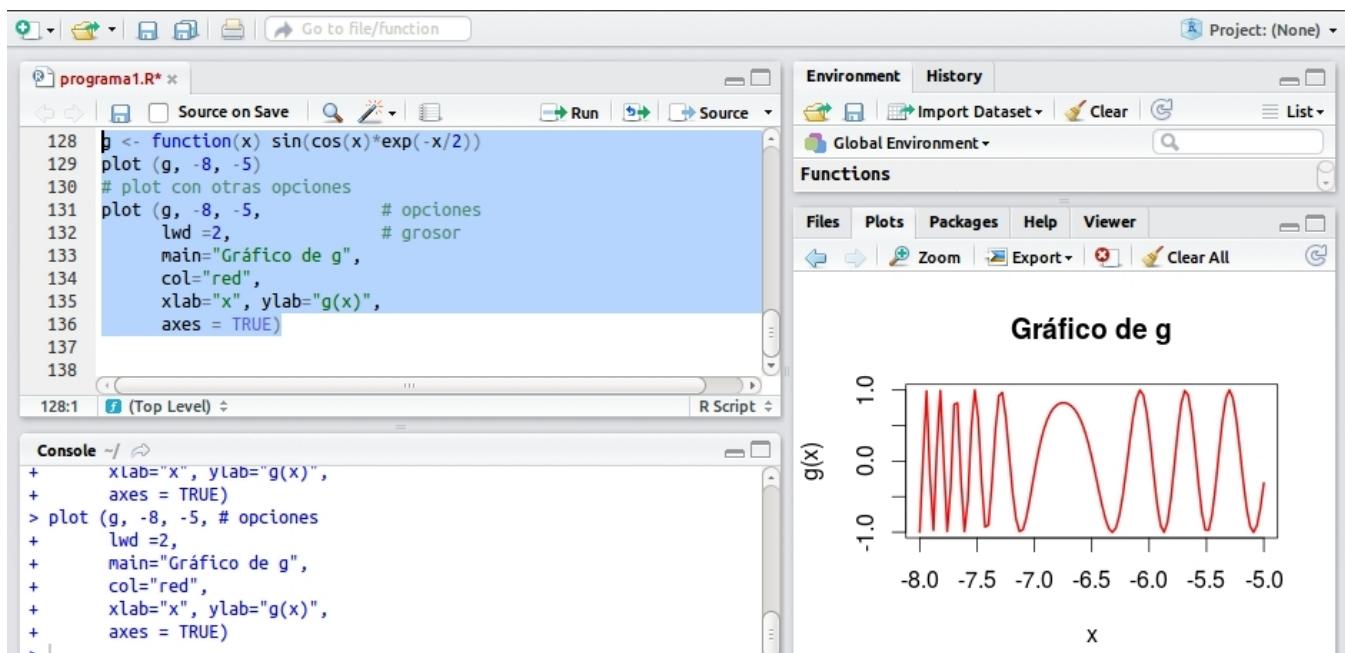
■ **Representación gráfica.** Para obtener un gráfico de una función f , se puede usar el comando **plot(f, a, b)** y también se pueden agregar varias opciones. Una lista de opciones de puede encontrar en

<http://stat.ethz.ch/R-manual/R-devel/library/graphics/html/par.html>

Por ejemplo, podemos hacer la representación gráfica de $g(x) = \sin(\cos(x)) * e^{-x/2}$ en $[-8, 5]$, y agregar opciones de grosor (**lwd**), color (**col**) y otras opciones.

■ Código R 1.11: Gráfica con opciones

```
g = function(x) sin(cos(x))*exp(-x/2)
plot(g, -8, -5, # rango
      lwd = 2, # grosor
      main = "Gráfico de g", col = "red", xlab = "x", ylab="g(x)", axes = TRUE, n = 100)
```



■ **curve()**. También se puede usar la función **curve()** para graficar introduciendo la fórmula directamente o por nombre. La sintaxis que por ahora vamos a usar, es la que sigue

```
curve(expr, a, b, n = 101, add = FALSE,
      type = "l", xname = "x", xlab = xname, ylab = NULL,
      log = NULL, xlim = NULL, ...)
```

Aquí `expr` es el nombre de una función de `x` o una fórmula en términos de `x`

■ Código R 1.12: `curve()`

```
# -- por expresión
curve(sin(x), -1, 1, col = 1)
curve(2*x^2+3*x+1, -1, 1, col = 1)
# -- por nombre
fun = function(x) x^2
curve(fun, -1, 1, col = 2, add = T) #add se usa para graficar junto al gráfico anterior
# -- Caso especial f(x) = x
curve((x), -1, 1, col = 2, add = T) # o curve(1*x,...),
```

1.6 Vectores

En el cálculo numérico con **R** se usa casi siempre vectores y matrices. Un vector es una lista ordenada de números, caracteres o valores lógicos; separados por comas.

■ Declarando vectores. Hay varias maneras de crear un vector: `c(...)` (c=combine), `seq(from, to, by)` (seq=sequence) y `rep(x, times)` (r=repeat).

La instrucción `n:m` crea una sucesión de números de n hasta m con paso $h = 1$. La instrucción `length(x)` nos devuelve la longitud del vector `x`

■ Código R 1.13: Declarando vectores

```
x = c(1.1, 1.2, 1.3, 1.4, 1.5) # x = (1.1,1.2,1.3,1.4,1.5)
x = 1:5                         # x = (1,2,3,4,5)
```

■ Código R 1.14: Declarando vectores con “seq”

```
x = seq(1,3, by =0.5)           # x = (1.0 1.5 2.0 2.5 3.0)
x = seq(5,1, by =-1)            # x = (5, 4, 3, 2, 1)
```

■ Código R 1.15: Declarando vectores con “rep”

```
x = rep(1, times = 5)          # x = (1, 1, 1, 1, 1)
length(x)                      # 5
```

■ Código R 1.16: Un vector con datos “no disponibles”

```
x = rep(NA, 5) # [1] NA NA NA NA NA
```

■ Código R 1.17: Muestra: Tirar una moneda limpia diez veces

```
x=c("cara", "cruz")
muestra = sample(x, size=10, replace = TRUE)
print(muestra)
# [1] "cruz" "cara" "cruz" "cruz" "cara" "cara" "cruz" "cruz" "cruz" "cara"
table(muestra)
# muestra
# cara cruz
# 4     6
```

■ **Operaciones algebraicas.** A los vectores les podemos aplicar las operaciones algebraicas usuales: sumar, restar, multiplicar y dividir, exponentiar (miembro a miembro), etc.

■ Código R 1.18: Operaciones con vectores

```
x = 1:5 # x = (1,2,3,4,5)
y = rep(0.5, times = length(x)) # y = (0.5,0.5,0.5,0.5,0.5)
x+y
# [1] 1.5 2.5 3.5 4.5 5.5
x*y
# [1] 0.5 1.0 1.5 2.0 2.5
x^y
# [1] 1.000000 1.414214 1.732051 2.000000 2.236068

1/(1:5) # = (1/1, 1/2, 1/3, 1/4, 1/5)
# [1] 1.00 0.50 0.3333333 0.25 0.2
```

■ **Reciclaje.** Cuando aplicamos operaciones algebraicas a vectores de distinta longitud, **R** automáticamente “repite” el vector más corto hasta que iguale la longitud del más grande

■ Código R 1.19:

```
c(1,2,3,4) + c(1,2) # = c(1,2,3,4)+c(1,2,1,2)
# [1] 2 4 4 6
```

Esto pasa también si tenemos vectores de longitud 1

■ Código R 1.20:

```
x = 1:5                      # x = (1,2,3,4,5)
2*x
# [1]  2  4  6  8 10
1/x^2
# [1] 1.0000000 0.2500000 0.1111111 0.0625000 0.0400000
x+3
# [1] 4 5 6 7 8
```

Ejemplo 1.2

- a.) El conjunto solución de la ecuación $\cos(5x) = 0$ es $\left\{ \frac{(2k+1)\pi}{6} \mid k \in \mathbb{Z} \right\}$. Las primeras soluciones, positivas y negativas, se obtienen dándole valores enteros a k .

Podemos implementar una función que nos de estas primeras soluciones.

```
sols1 = function(n,m){
  valoresk = n:m                # "valoresk" es un vector
  (2*valoresk+1)*pi/6          # la salida es un vector
}
#Llamada al programa: soluciones desde k=-5 hasta k=5
sols1(-5,5)
# [1] -4.7123890 -3.6651914 -2.6179939 -1.5707963 -0.5235988  0.5235988
# [7]  1.5707963  2.6179939  3.6651914  4.7123890  5.7595865
```

- b.) Una partición del intervalo $[a, b]$ en n subintervalos igualmente espaciados es una colección de *nodos* $\{a = x_0, x_1, x_2, \dots, x_n = b\}$ donde

$$h = \frac{b-a}{n} \quad \text{y} \quad x_i = x_0 + h \cdot i$$

Podemos implementar un función que dados a , b y n , nos devuelva la partición.

```
particion = function(a,b,n){
  h=(b-a)/n
  i = 0:n                  # "i" es un vector
  a + h*i                  # la salida es un vector
}
#Llamada al programa: Una partición de [0,1] en 10 subintervalos
```

```
particion(0,1,10)
# [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

■ **Funciones que aceptan vectores como argumentos.** Algunas funciones se pueden aplicar sobre vectores, por ejemplo **sum()**, **prod()**, **max()**, **min()**, **sqrt()**, **mean()**, **var()**, **sort()**

■ **Código R 1.21:**

```
x = 1:5                      # x = (1,2,3,4,5)
sqrt(x)
# [1] 1.000000 1.414214 1.732051 2.000000 2.236068
sum(x)  #1+2+3+4+5
# [1] 15
prod(x) #1*2*3*4*5
# [1] 120
mean(x) #promedio
# [1] 3
var(x) #varianza
#[1] 2.5
```

Ejemplo 1.3

Es conocido que $S = \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6} \approx 1.64493$. Podemos aproximar la suma de la serie con sumas parciales, por ejemplo

$$S \approx S_k = \sum_{i=1}^k \frac{1}{i^2}$$

Para implementar esta suma parcial en **R**, usamos operaciones con vectores,

$$\begin{aligned}\sum_{i=1}^k \frac{1}{i^2} &= 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{k^2} \\ &= \text{sum}(1, 1/2^2, 1/3^2, \dots, 1/k^2)\end{aligned}$$

Si $x = 1:k$, entonces

$$= \text{sum}(x^2)$$

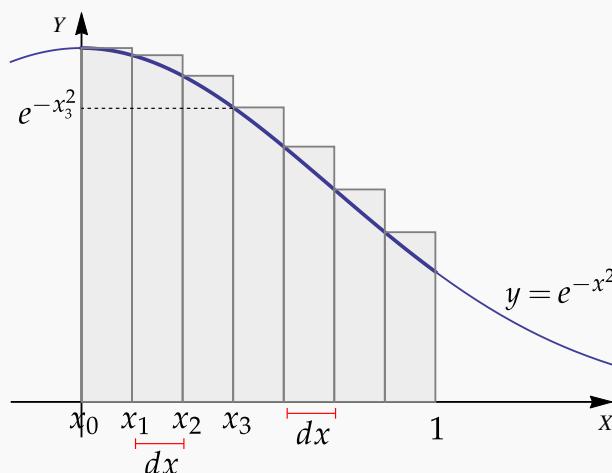
■ Código R 1.22:

```
sumaparcial1 = function(k){ x = 1:k
                           sum(1/x^2)
}
# Llamada al programa
sumaparcial1(100000)
# [1] 1.644924
```

Ejemplo 1.4

También podemos armar de manera simple la aproximación de una integral (en el sentido de Riemann). La primitiva $\int_0^1 e^{-x^2} dx$ no se puede calcular con los métodos usuales de cálculo.

Una aproximación sencilla es la suma de las áreas de los rectángulos bajo la curva, de base dx y altura $e^{-x_i^2}$,



Área del rectángulo i -ésimo = base · altura = $dx \cdot e^{-x_i^2}$

$$\int_0^1 e^{-x^2} dx \approx e^{-x_0^2} \cdot dx + e^{-x_1^2} \cdot dx + e^{-x_2^2} \cdot dx + \dots + e^{-x_n^2} \cdot dx$$

$$\approx \text{sum}(e^{-x_0^2}, e^{-x_1^2}, e^{-x_2^2}, \dots, e^{-x_n^2}) * dx$$

■ Código R 1.23:

```
area1 = function(dx){
  xi = seq(0,1, by = dx) # = (0, dx, 2dx, ..., 1)
  sum(exp(-xi^2))*dx
}
# Llamada a la función
area1(1/10000000)
# [1] 0.7468242
```

■ **Acceder a las entradas y listas de componentes.** La entrada i -ésima del vector x es $x[i]$. El lenguaje R ejecuta operaciones del tipo $x[\text{op}]$ donde op es un operación lógica válida.

■ Código R 1.24:

```
x=c(1.1,1.2,1.3,1.4,1.5) # declarar un vector x
x[2]                      # entrada 2 de x
# [1] 1.2
x[3];x[6]                  # entradas 3 y 6 de x
# [1] 1.3
# [1] NA                     # no hay entrada 6, se trata como un dato "perdido"
x[1:3]; x[c(2,4,5)]       # sublistas
# [1] 1.1 1.2 1.3
# [1] 1.2 1.4 1.5
x[4]=2.4                   # cambiar una entrada
x
# [1] 1.1 1.2 1.3 2.4 1.5
x[-3]                      # remover el elemento 3
# [1] 1.1 1.2 2.4 1.5
x = c( x[1:3],16, x[4:5] ) # Insertar 16 entre la entrada 3 y 4
# [1] 1.1 1.2 1.3 16.0 2.4 1.5
```

En el código que sigue se muestra algunos ejemplos en los que se aplica operaciones de tipo lógico en el vector x

■ Código R 1.25:

```

x = 1:5
set.seed(234)           # "semilla"
y = sample(1:10, 5)      # y = (8, 10, 1, 6, 9)
z = x[x<2.5]; z        # declara z con una regla: los x_i < 2.5
# [1] 1 2
e = x[x==y[3]]; e       # declara e con una regla: los x_i iguales a y[3]
# [1] 1

```

Ejemplo 1.5

Si tenemos un conjunto de datos numéricos $x = c(x_1, x_2, \dots, x_n)$ (una muestra) entonces su media (o promedio) es $\text{sum}(x)/\text{length}(x)$, en R esta se calcula con $\text{mean}(x)$.

Sin embargo si la muestra tiene datos atípicos, entonces la media se distorsiona. En estos casos se usan un par de métodos alternativos para calcular la media: La “media podada” y la “media Winsorizada”. Lo que hacen es ordenar los datos de menor a mayor para que los puntos atípicos queden en la rama inferior o en la rama superior y se hace una poda de estas ramas o se sustituyen estas ramas con valores posiblemente más sensatos. Por ejemplo,

```

# Muestra
x = c(1, 87, 59, 05, 67, 65, 68, 56, 78, 80, 67, 59, 800, 100, 99, 1000 )
# ordenamos
xs = sort(x); xs
# [1] 1 5 56 59 59 65 67 67 68 78 80 87 99 100 800 1000

```

Ahora se observan los datos aparentemente atípicos **1,5** y **800,1000**.

Supongamos que tenemos un vector x de datos (una muestra) ordenados de menor a mayor, $x = (x_1, x_2, \dots, x_n)$. La k -ésima muestra podada es

$$\bar{x}_k = \frac{x_{k+1} + x_{k+2} + \dots + x_{n-k}}{n-2k}$$

Es decir, la k -ésima muestra podada es la media de los datos que quedan al descartar los primeros y los últimos k datos.

■ Código R 1.26: Función media podada k-ésima

```
mediaP = function(x,k){
```

```

n = length(x)
xs = sort(x)
xt = xs[(k+1):(n-k)] #eliminar k primeros y k últimos
mean(xt)
}

```

La k -ésima media Winsorizada en vez de descartar los k primeros y los k últimos datos, los sustituye, cada uno de ellos, por los datos x_{k+1} y x_{n-k} .

$$\bar{w}_k = \frac{(k+1)x_{k+1} + x_{k+2} + \dots + x_{n-k-1} + (k+1)x_{n-k}}{n}$$

Por ejemplo, si $x=(1,2,3,4,5,6,7,8,9,10)$ entonces $\bar{w}_3 = (4+4+4+ 4+5+6+7 +7+7+7)/10$

■ Código R 1.27:k-ésima media Winsorisada

```

mediaW = function(x, k) {
  x = sort(x)
  n = length(x)
  x[1:k] = x[k+1]
  x[(n-k+1):n] = x[n-k]
  return(mean(x))
}

```

Veamos las medias aplicadas a una muestra,

■ Código R 1.28:Probando las medias

```

# ---
x = c( 8.2, 51.4, 39.02, 90.5, 44.69, 83.6, 73.76, 81.1, 38.81, 68.51)
k= 2
cat(mean(x)," ", mediaP(x,k)," ", mediaW(x,k))
# 57.959      68.77833      59.872

# --- Introducir un valor atípico
xat = x
xat[1] = 1000
cat(mean(xat)," ", mediaP(xat,k)," ", mediaW(xat,k))
# 157.139      68.77833      65.964

```

■ **Otras funciones.** Hay muchas operaciones y funciones que aplican sobre vectores, en la tabla que sigue se enumeran algunas de ellas.

sum(x) :	suma de los elementos de x
prod(x) :	producto de los elementos de x
max(x) :	valor máximo en el objeto x
min(x) :	valor mínimo en el objeto x
which.max(x) :	devuelve el índice del elemento máximo de x
which.min(x) :	devuelve el índice del elemento mínimo de x
range(x) :	rango de x, es decir, c(min(x), max(x))
length(x) :	número de elementos en x
mean(x) :	promedio de los elementos de x
median(x) :	mediana de los elementos de x
round(x, n) :	redondea los elementos de x a <i>n</i> cifras decimales
rev(x) :	invierte el orden de los elementos en x
sort(x) :	ordena los elementos de x en orden ascendente
cumsum(x) :	un vector en el que el elemento <i>i</i> es la suma acumulada hasta <i>i</i>
cumprod(x) :	igual que el anterior pero para el producto
cummin(x) :	igual que el anterior pero para el mínimo
cummax(x) :	igual que el anterior pero para el máximo
match(x, y) :	devuelve un vector de igual longitud que x con los elementos de x que están en y
which(x==a) :	devuelve un vector con los índices de x si la operación es TRUE. El argumento de esta función puede cambiar si es una expresión de tipo lógico

Ejemplo 1.6

Usando la función **cumprod**

```

n=10
cumprod(2:n) # = (2!, 3!, 4!, ..., n!)
#[1]      2       6      24     120      720     5040    40320   362880 3628800
x=1; k=10
x/(1:k)      # = (x/1, x/2, x/3, ..., x/k)
cumprod(x/(1:k)) # = (x/1!, x^2/2!, x^3/3!, ..., x^k/k!)

```

Ejemplo 1.7

Se sabe que $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. En esta fórmula se usa como un convenio que $0^0 = 1$. Consideremos las sumas parciales $\sum_{n=0}^k \frac{x^n}{n!}$,

Si $k = 20$ entonces $e^x \approx \sum_{n=0}^{20} \frac{x^n}{n!}$, es decir, $e^x \approx 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{20}}{20!}$. En particular,

$$e^1 \approx 1 + \frac{1}{1} + \frac{1^2}{2!} + \frac{1^3}{3!} + \dots + \frac{1^{20}}{20!}$$

$$e^{-10} \approx 1 + \frac{-10}{1} + \frac{(-10)^2}{2!} + \frac{(-10)^3}{3!} + \dots + \frac{(-10)^{20}}{20!}$$

En **R** podemos calcular las sumas parciales en formato vectorizado. Observemos:

$$\begin{aligned} e^x &\approx 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^k}{k!} \\ &\approx 1 + \text{sum}\left(\frac{x}{1}, \frac{x^2}{2!}, \frac{x^3}{3!}, \dots, \frac{x^k}{k!}\right) \\ &\approx 1 + \text{sum}\left(\frac{x}{1}, \frac{x}{1} \cdot \frac{x}{2}, \frac{x}{1} \cdot \frac{x}{2} \cdot \frac{x}{3}, \dots, \frac{x}{1} \cdot \frac{x}{2} \cdots \frac{x}{k}\right) \\ \text{como } \frac{x}{1:k} &= \left(\frac{x}{1}, \frac{x}{2}, \frac{x}{3}, \dots, \frac{x}{k}\right), \text{ entonces} \\ &\approx 1 + \text{sum}(\text{cumprod}(\frac{x}{1:k})) \end{aligned}$$

```
funexp = function(x, n) 1 + sum(cumprod(x/1:n)) #=sum(x/1, x/1*x/2, x/1*x/2*x/3,...)

# --- Comparación con exp(x)
c(funexp(1,20), exp(1))
[1] 2.718282    2.718282

c(funexp(-10,20), exp(-10))
#[1] 1.339687e+01 4.539993e-05 (!)

# mejor e^-10 = 1/e^10
c(1/funexp(10,20), exp(-10))
```

```
#[1] 4.547215e-05 4.539993e-05
```

Esta implementación solo es eficiente para números positivos

- **La función `sapply()`.** La función `sapply(x, FUN)` aplica (en su manera más simple) la función `FUN` a cada componente del vector `x` y devuelve un vector del mismo tamaño que `x`

■ Código R 1.29: Usando `sapply()`

```
fc = function(x) x^2
x = 1:5
sapply(x, FUN = fc)
# [1] 1 4 9 16 25
```

1

Ejercicios

1. Implementar una función `ps(k)` que recibe un natural k y devuelve la lista de números $\{6i + 1 : i = 0, 1, 2, \dots, k\} = \{1, 7, 13, \dots, 6k + 1\}$.

```
ps = function(k){
    #...
}
```

Ejemplos de salidas:

```
ps(3) devuelve 1 7 13 19
ps(5) devuelve 1 7 13 19 25 31
```

2. El conjunto solución de la ecuación $\sin(5x) = 0$ es $\left\{\frac{k\pi}{5} \text{ con } k \in \mathbb{Z}\right\}$. Implementar una función `sols(n,m)` que nos devuelva las soluciones desde $k = n$ hasta $k = m$.

```
sols = function(n,m){
    #...
}
```

Ejemplos de salidas:

```
sols(-2,3) devuelve -1.2566371 -0.6283185 0.0000000 0.6283185 1.2566371 1.8849556
```

```
sols(-4,1) devuelve -2.5132741 -1.8849556 -1.2566371 -0.6283185 0.0000000 0.6283185
```

3. Dada una partición $\{x_0, x_1, \dots, x_n\}$ del intervalo $[a, b]$ igualmente espaciada, con paso $h = (b - a)/n$, implementar dos funciones **paresx(a,b,n)** e **imparesx(a,b,n)**.

La primera **paresx(a,b,n)** devuelve los nodos de subíndice par, sin tomar el primero ni el último, es decir, devuelve $x_2, x_4, x_6, \dots, x_s$. El último nodo sería x_{n-2} si n es par y x_{n-1} si n es impar.

```
paresx = function(a,b,n){
  h=(b-a)/n
  #...
}
```

La segunda función **imparesx(a,b,n)** devuelve los nodos de subíndice impar, sin tomar el primero ni el último, es decir, devuelve $x_1, x_3, x_5, \dots, x_s$. El último nodo sería x_{n-1} si n es par y x_{n-2} si n es impar.

```
imparesx = function(a,b,n){
  #...
}
```

Ejemplos de salidas:

```
particion(0,1,5)
#[1] 0.0 0.2 0.4 0.6 0.8 1.0
paresx(0,1, 5)
#[1] 0.4 0.8
imparesx(0,1, 5)
#[1] 0.2 0.6
```

```
particion(1,12,6)
#[1] 1.000000 2.833333 4.666667 6.500000 8.333333 10.166667 12.000000
paresx(1,12,6)
#[1] 4.666667 8.333333
imparesx(1,12,6)
#[1] 2.833333 6.500000 10.166667
```

4. Considera la serie $\sum_{n=1}^{\infty} \frac{1}{n^3 \sin^2(n)}$. Implementar una función **sparcial(k)** = $\sum_{n=1}^k \frac{1}{n^3 \sin^2(n)}$ para calcular la k -ésima suma parcial de esta serie. El código inicia así:

```
sparcial = function(k){
    #...
}
```

Ejemplos de salidas:

```
sparcial(1)
# [1] 1.412283
sparcial(1000)
# [1] 30.17479
sparcial(10000)
# [1] 30.31451
sparcial(100000)
# [1] 30.31452
```

5. Sea **vx=c(x0, x1, ..., xn)** y $x \in \mathbb{R}$. Se define la función $\ell(vx, x) = (x - x_0)(x - x_1) \cdots (x - x_n)$. Implementar la función **ele(vx, x)**

```
ele = function(vx,x){
    #...
}
```

Notar que si que tenemos un vector **vx** de datos $x_0, x_1, x_2, \dots, x_n$; la entrada x_0 corresponde a **vx[1]**, la entrada x_1 corresponde a **vx[2]** y, en general, la entrada x_k corresponde a **vx[k+1]**. De esta manera **k** varía de **1** hasta **length(vx)-1**. El código inicia así:

Ejemplos de salidas:

```
xx=c(1,2,3,4,5)
ele(xx, 0)
# [1] -120
ele(xx, 1)
# [1] 0
```

```

ele(xx, 0.5)
# [1] -29.53125
ele(xx, -0.5)
# [1] -324.8438
ele(xx, 0.5)
# [1] -29.53125
ele(xx, 6.5)
# [1] 324.8438

```

6. Sea $x = c(x_0, x_1, \dots, x_n)$. Se define la función $w(x, k) = \prod_{\substack{i=0 \\ i \neq k}}^n (x_k - x_i)$ es decir,

$$w(x, k) = (x_k - x_0) \cdots (x_k - x_{k-1}) \curvearrowright (x_k - x_{k+1}) \cdots (x_k - x_n)$$

Implementar la función **w(vx, k)**

```

w = function(vx,k){
  #...
}

```

Ejemplos de salidas:

```

xx=c(0,3,4,4)
w(xx, 0)
# [1] -48
w(xx, 1)
# [1] 3
w(xx, 2)
# [1] 0
w(xx, 3)
# [1] 0

```

7. Si f es una función continua y mayor o igual a cero en $[a, b]$, entonces una aproximación con n rectángulos, de la integral $\int_a^b f(x) dx$, es

$$\int_a^b f(x) dx \approx \sum_{i=0}^n f(x_i) \Delta x \quad \text{donde} \quad \Delta x = \frac{b-a}{n} \quad \text{y} \quad x_i = a + i \cdot \Delta x, \quad i = 0, 1, 2, \dots, n$$

Implementar la función **approxRiemann(f,a,b,n)**= $\sum_{i=0}^n f(x_i)\Delta x$

Ejemplos de salidas:

```
fun = function(x) sin(x)
approxRiemann(fun,0,1,100)
# [1] 0.4639012

g = function(x) sin(x)/x
approxRiemann(g,1,pi,1000)
# [1] 0.906755
```

1.7 Paquetes

Además de los paquetes estándar que vienen con **R**, se pueden instalar paquetes adicionales para aumentar la cantidad de funciones, conjuntos de datos y librerías con algún propósito.

■ **Paquete `pracma`:** Por ejemplo, el producto punto y las normas ya están implementadas en el paquete **pracma** (“Practical Numerical Math Functions”, <http://cran.r-project.org/web/packages/pracma/pracma.pdf>). Podríamos instalar este paquete en nuestra copia local de **R** (una sola vez) y luego usarlo (cargar las funciones del paquete) con la función **require()**

■ Código R 1.30: Paquete “pracma”

```
install.packages("pracma") # Se instala solo una vez
require(pracma)           # "convocar" el paquete instalado
```

Ahora ya podemos usar las funciones del paquete **pracma**. Las funciones aparecen en la documentación del paquete.

Por ejemplo, dos funciones son **dot(u,v)** (producto punto $u \cdot v$) y **cross(u,v)** (producto cruz $u \times v$)

■ Código R 1.31: funciones “dot”y “cross” del paquete “pracma”

```
x=c(2,3,4)
y=c(1,1,1)
dot(x,y)
# [1] 9
cross(x,y)
# [1] -1  2 -1
```

■ **require() y library()**. Para usar un paquete se usa **require(paquete)** o también **library(paquete)**, ambos cargan el paquete de nombre “paquete”. En realidad **requiere()** está hecho para cargar dentro de las funciones, devuelve **TRUE** si el paquete está disponible y **FALSE** sino.

■ Código R 1.32: require() y library()

```
pruebaA <- library("abc"); pruebaA
# Error en library("abc") : there is no package called 'abc'
pruebaB <- require("abc"); pruebaB
# Loading required package: abc
# Mensajes de aviso perdidos
# In library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE, :
#   there is no package called 'abc'
# [1] FALSE
```

Otros paquetes útiles son:

Matrix: Ofrece clases y métodos para operar con matrices densas y ralas

optR: Utiliza métodos elementales del álgebra lineal (Gauss, LU, CGM, Cholesky) para resolver sistemas lineales

R.matlab: Herramientas para leer y editar archivos .mat de Matlab

Rmpfr: Para trabajar con números con precisión “infinita” en vez de doble precisión (a lo sumo 16 dígitos).

Una lista de paquetes útiles en métodos numéricos se puede ver en “CRAN Task View: Numerical Mathematics” en <http://cran.r-project.org/web/views/NumericalMathematics.html>

También se puede consultar “Quick list of useful R packages” en <https://support.rstudio.com/hc/en-us/articles/201057987-Quick-list-of-useful-R-packages>

1.8 Matrices y arreglos

Las matrices son arreglos bidimensionales y, por defecto, se declaran “por columnas”. En el siguiente código se muestra como acceder a las entradas (i, j) y algunas operaciones con matrices.

■ **Declarando una matriz**: La sintaxis para declarar una matriz es

```
A =matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

El parámetro **data**, para este curso, es una matriz en formato de vector. Las opciones **nrow** y **ncol** son el número de filas y el número de columnas (una especificación de cómo se debe “armar” la matriz). La opción **byrow=TRUE**

indica que el vector `data` se debe leer por filas. La opción `dimnames` se usa para etiquetar filas y/o columnas con una lista: `dimnames = list(...,...)`.

Además hay algunas maneras cortas de declarar matrices especiales. Por ejemplo, las funciones `cbind()` (combine column) y `rbind()` (combine row) se usa para combinar vectores y matrices por columnas o por filas.

■ Código R 1.33: Declarar matrices

```
# --- Matriz nula 3x3
A = matrix(rep(0,9), nrow = 3, ncol= 3); A
#      [,1] [,2] [,3]
#[1,]     0     0     0
#[2,]     0     0     0
#[3,]     0     0     0
# --- Declarar una matriz por filas (el default es "por columnas")
B = matrix(c(1,2,3,
            5,6,7), nrow = 2, byrow=T); B
#      [,1] [,2] [,3]
#[1,]     1     2     3
#[2,]     5     6     7

# --- Declarar primero las columnas "x", "y" y "z"
x = 1:3; y = seq(1,2, by = 0.5); z = rep(8, 3) ; x; y; z
#[1] 1 2 3
#[1] 1.0 1.5 2.0
#[1] 8 8 8

# --- Combinar "x", "y" y "z" para formar una matriz
C = matrix(c(x,y,z), nrow = length(x)); C # ncol no es necesario declararlo
#      [,1] [,2] [,3]
#[1,]     1   1.0     8
#[2,]     2   1.5     8
#[3,]     3   2.0     8

# --- Construir la matriz por filas (rbind) o por columnas (cbind)
xi = seq(1,2, by 0.1); yi = seq(5,10, by = 0.5)
rbind(xi,yi)
# [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
#xi    1  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9    2
#yi    5  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5   10
cbind(xi,yi)
#      xi    yi
# [1,] 1.0  5.0
# [2,] 1.1  5.5
```

```
# [3,] 1.2 6.0
# [4,] 1.3 6.5
# [5,] 1.4 7.0
# [6,] 1.5 7.5
# [7,] 1.6 8.0
# [8,] 1.7 8.5
# [9,] 1.8 9.0
#[10,] 1.9 9.5
#[11,] 2.0 10.0
```

■ **Extraer/modificar la diagonal.** Si A es una matriz cuadrada, el comando `diag(A)` construye una matriz diagonal o también extrae la diagonal de una matriz. La instrucción `I=diag(1, n)` asigna a `I` la matriz identidad $n \times n$ y la instrucción `D = diag(diag(A))` asigna a `D` una “matriz diagonal” con la diagonal de la matriz A .

■ Código R 1.34: Función “`diag()`”

```
# --- construir una matriz diagonal
I = diag(c(3,1,3)); I

#      [,1] [,2] [,3]
#[1,]    3    0    0
#[2,]    0    1    0
#[3,]    0    0    3

# --- Extraer la diagonal de la matriz I
diag(I)
[1] 3 1 3
# --- Matriz diagonal nxn
n = 3
I = diag(1, n);
#      [,1] [,2] [,3]
#[1,]    1    0    0
#[2,]    0    1    0
#[3,]    0    0    1
# --- Matriz diagonal, con la diagonal de A
D = diag(diag(A))

# --- Matriz nxm, con 1's en las entradas a_{ii}: diag(1, n, m)
J = diag(1, 3, 4); J
[,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
```

■ **Operador “outer()”.** El operador `outer()` crea una nueva matriz $m \times n$ combinando vectores de dimensiones m y n , de acuerdo a la regla `FUN`. Por defecto, `FUN = "*"`.

Por ejemplo, si definimos una función `f = function(x,y){...}`, entonces `outer` construye la matriz en azul,

<code>FUN = "f"</code>	y_1	y_2	...	y_m	
x_1	<code>f(x₁,y₁)</code>	<code>f(x₁,y₂)</code>	...	<code>f(x₁,y_m)</code>	
x_2	<code>f(x₂,y₁)</code>	<code>f(x₂,y₂)</code>	...	<code>f(x₂,y_m)</code>	= <code>outer(x,y,FUN ="f")</code>
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
x_n	<code>f(x_n,y₁)</code>	<code>f(x_n,y₂)</code>	...	<code>f(x_n,y_m)</code>	

■ Código R 1.35: Función “outer()”

```
x = 4:6; y = 1:3
outer(x,y) # Fun = "*" por defecto
%, [,1] [,2] [,3]
%[1,] 4 8 12
%[2,] 5 10 15
%[3,] 6 12 18
outer(x,y, FUN = "+")
%, [,1] [,2] [,3]
%[1,] 5 6 7
%[2,] 6 7 8
%[3,] 7 8 9
```

■ **Entradas y bloques.** Podemos acceder a la fila i con la instrucción `A[i,]` y a la columna j con la instrucción `A[, j]`. Se puede declarar submatrices `B` de `A` con la instrucción `B=A[vector1, vector2]`. La instrucción `B=A` hace una copia (independiente) de `A`

■ Código R 1.36: Entradas, filas y columnas de una matriz

```
B = matrix(c( 1, 1 ,8,
            2, 0, 8,
            3, 2, 8), nrow = 3, byrow=TRUE); B
#, [,1] [,2] [,3]
#[1,] 1 1 8
#[2,] 2 0 8
#[3,] 3 2 8
# --- Entrada (2,3)
B[2, 3]
#[1] 8
```

```
# --- fila 3
B[3,]
#[1] 3 2 8
# --- columna 2
B[,2]
#[1] 1 0 2
# --- bloque de B
B[1:2,c(2,3)]
# [,1] [,2]
#[1,] 1 8
#[2,] 0 8
```

■ **Operaciones de fila.** Para cambiar la fila i y la fila j se usa la instrucción

$$A[c(i,j),] = A[c(j,i),]$$

Las operaciones usuales de fila $\alpha F_i + \beta F_j$ sobre la matriz A se hacen de manera natural

$$A[j,] = alpha*A[i,] + beta*A[j,]$$

■ Código R 1.37: Operaciones de fila

```
A = matrix(c( 1, 1 ,8,
             2, 0, 8,
             3, 2, 8), nrow = 3, byrow=TRUE); A
# [,1] [,2] [,3]
#[1,] 1 1 8
#[2,] 2 0 8
#[3,] 3 2 8
A[c(1,3), ] = A[c(3,1), ]           # Cambio F1, F3
# [,1] [,2] [,3]
#[1,] 3 2 8
#[2,] 2 0 8
#[3,] 1 1 8

A[2, ] = A[2, ] - A[2,1]/A[1,1]*A[1, ] # F2 - a_{21}/a_{11}*F1
# [,1] [,2] [,3]
#[1,] 3 2.000000 8.000000
#[2,] 0 -1.333333 2.666667
#[3,] 1 1.000000 8.000000
```

■ **Pivotes.** A veces es necesario determinar el índice de la entrada más grande, en valor absoluto, de una fila. Para esto podemos usar la función **which.max(x)**: Esta función devuelve el índice de la entrada más grande, del vector x , por ejemplo

■ **Código R 1.38: Índice de la entrada más grande de un vector**

```
x = c(2, -6, 7, 8, 0.1,-8.5, 3, -7, 3)
which.max(x)      # max(x) = x[4]
# [1] 4
which.max(abs(x)) # max(abs(x)) = abs(x[6])
# [1] 6
```

Ahora, el *índice* de la entrada más grande, en valor absoluto, de la fila **k** de la matriz **A** es **which.max(abs(A[k, 1])**

Ejercicio 1.1 Considere el problema: Determinar el *índice* de la entrada más grande, en valor absoluto, de la fila **k** de la matriz **A** que está por debajo de la diagonal. Explique por qué la solución a este problema es

which.max(abs(M[k:n,k]))+ k-1

■ **Matriz triangular superior y triangular inferior.** En algunos algoritmos se requiere extraer las matrices triangular inferior y estrictamente inferior (y superior respectivamente). Esto se puede hacer usando un operador lógico **A[op]** donde **op** es una operación lógica. Por ejemplo,

- a.) **nrow(A)** y **ncol(A)** devuelven el número de filas y el número de columnas de **A**, respectivamente.

Por otro lado **row(A)** devuelve una matriz de enteros con las mismas dimensiones que **A** y cuya entrada a_{ij} es igual a **i**.

Por otro lado **col(A)** devuelve una matriz de enteros con las mismas dimensiones que **A** y cuya entrada a_{ij} es igual a **j**.

- b.) la matriz triangular inferior de **A** tiene ceros arriba de la diagonal: **A[col(A)>= row(A)+1] = 0**

- c.) la matriz triangular estrictamente inferior de **A** tiene ceros arriba y en la diagonal:

$$\mathbf{A}[col(A)> row(A)+1] = 0$$

- d.) la matriz triangular superior: **A[col(A)<= row(A)-1] = 0**

- e.) la matriz triangular estrictamente inferior de **A** tiene ceros arriba y en la diagonal:

$$\mathbf{A}[col(A)<= row(A)] = 0.$$

■ **Código R 1.39:**

```
A = matrix(c( 1, 1 ,8, 5,
            2, 0, 8, 9,
            3, 2, 8,  3,
```

```

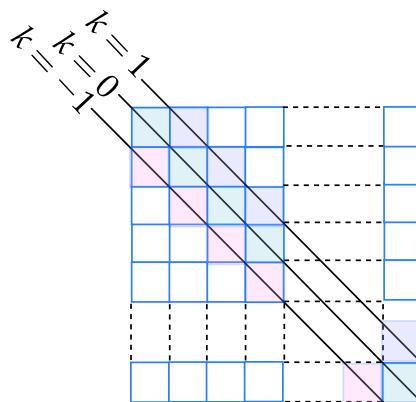
2, 6, 5, 1), nrow = 4, byrow=TRUE)

LA = A
LA[col(LA) >= row(LA)+1] = 0; LA
# [,1] [,2] [,3] [,4]
#[1,] 1 0 0 0
#[2,] 2 0 0 0
#[3,] 3 2 8 0
#[4,] 2 6 5 1

LA = A
LA[col(LA) >= row(LA)] = 0; LA
# [,1] [,2] [,3] [,4]
#[1,] 0 0 0 0
#[2,] 2 0 0 0
#[3,] 3 2 0 0
#[4,] 2 6 5 0

```

■ **Usando el paquete Matrix.** El paquete **Matrix** viene con implementaciones de funciones matriciales y otros algoritmos. Hay tres funciones muy útiles: **tril(X,k)**, y **triu(X,k)**. La función **triu(X,k)** devuelven los elementos de X sobre la diagonal principal y arriba de esta diagonal hasta la k -ésima diagonal si $k > 0$ y desde la k -ésima diagonal hacia arriba, si $k < 0$. La función **tril(X,k)** hace algo similar pero hacia abajo de la k -ésima diagonal.



El siguiente ejemplo fue tomado de la documentación del paquete,

■ **Código R 1.40:**

```

mm = matrix(rep(1, 9), nrow=3, byrow=T); mm
# [,1] [,2] [,3]
# [1,] 1 1 1
# [2,] 1 1 1
# [3,] 1 1 1
tril(mm)      # lower triangle
# 3 x 3 Matrix of class "dtrMatrix"
# [,1] [,2] [,3]

```

```

# [1,]    1    .    .
# [2,]    1    1    .
# [3,]    1    1    1
triu(mm)      # upper triangle
# 3 x 3 Matrix of class "dtrMatrix"
# [,1] [,2] [,3]
# [1,]    1    1    1
# [2,]    .    1    1
# [3,]    .    .    1
tril(mm, -1)  # strict lower triangle
# 3 x 3 Matrix of class "dtrMatrix"
# [,1] [,2] [,3]
# [1,]    0    .    .
# [2,]    1    0    .
# [3,]    1    1    0
tril(mm, 1)   # strict lower triangle
# 3 x 3 Matrix of class "dgeMatrix"
# [,1] [,2] [,3]
# [1,]    1    1    0
# [2,]    1    1    1
# [3,]    1    1    1
triu(mm, 1)   # strict upper triangle
# 3 x 3 Matrix of class "dtrMatrix"
# [,1] [,2] [,3]
# [1,]    0    1    1
# [2,]    .    0    1
# [3,]    .    .    0
triu(mm, -1)  # strict upper triangle
# 3 x 3 Matrix of class "dgeMatrix"
# [,1] [,2] [,3]
# [1,]    1    1    1
# [2,]    1    1    1
# [3,]    0    1    1

```

1.8.1 Operaciones con matrices

Las operaciones con matrices son similares a las que ya vimos con vectores. Habrá que tener cuidados con las dimensiones, por ejemplo las suma y resta de matrices solo es posible si tienen el mismo orden y $A * B$ es una multiplicación miembro a miembro mientras que la multiplicación matricial ordinaria es $A_{n \times k} \cdot B_{k \times n} = A \% * \% B$.

■ Código R 1.41: Operaciones con matrices

```
A = matrix(1:9, nrow=3); A      # Por columnas
```

```
B = matrix(rep(1,9), nrow=3); B
#[,1] [,2] [,3]
#[1,]    1    4    7
#[2,]    2    5    8
#[3,]    3    6    9
#[,1] [,2] [,3]
#[1,]    1    1    1
#[2,]    1    1    1
#[3,]    1    1    1
# --- Suma
A+B
#[,1] [,2] [,3]
#[1,]    2    5    8
#[2,]    3    6    9
#[3,]    4    7   10
# --- Producto miembro a miembro
A*B
#[,1] [,2] [,3]
#[1,]    1    4    7
#[2,]    2    5    8
#[3,]    3    6    9
# --- multiplicación matricial
A%*% B
#[,1] [,2] [,3]
#[1,]   12   12   12
#[2,]   15   15   15
#[3,]   18   18   18
A^2 # No es A por A!
#[,1] [,2] [,3]
#[1,]    1   16   49
#[2,]    4   25   64
#[3,]    9   36   81
A%*%A
#[,1] [,2] [,3]
#[1,]   30   66  102
#[2,]   36   81  126
#[3,]   42   96  150
# --- Restar 2 a cada A[i,j]
A - 2
#[,1] [,2] [,3]
#[1,]   -1    2    5
#[2,]    0    3    6
#[3,]    1    4    7
```

```

# --- Producto escalar
3*A
#     [,1] [,2] [,3]
#[1,]    3   12   21
#[2,]    6   15   24
#[3,]    9   18   27
# --- Transpuesta
t(A)
#     [,1] [,2] [,3]
#[1,]    1    2    3
#[2,]    4    5    6
#[3,]    7    8    9
# --- Determinante
det(A)
# [1] 0
# --- Inversas
C = A - diag(1,3)
det(C)
# [1] 32
# Inversa de C existe
solve(C)
#     [,1]     [,2]     [,3]
#[1,] -0.50  0.31250  0.1250
#[2,]  0.25 -0.65625  0.4375
#[3,]  0.00  0.37500 -0.2500

```

1.9 Función apply()

La función **apply(X, MARGIN, FUN)** retorna un vector (o un arreglo o una lista) con valores obtenidos al aplicar una función **FUN** a las filas o columnas (o ambas) de **X**. “**MARGIN**” es un índice. Si **X** es una matriz, **MARGIN = 1** indica que la operación se aplica a las filas mientras que **MARGIN = 2** indica que la operación se aplica a las columnas. **MARGIN = c(1,2)** indica que la función se aplica a ambas filas y columnas, es decir, a todos los elementos de la matriz. **FUN** es la función que se aplica y “...” se usa para argumentos opcionales de **FUN**

■ Código R 1.42: Usando apply()

```

A = matrix(1:9, nrow=3); A
#     [,1] [,2] [,3]
#[1,]    1    4    7
#[2,]    2    5    8
#[3,]    3    6    9
filas.suma <- apply(A, 1, sum)      #filas.sum = vector con las sumas de las filas

```

```
col.suma <- apply(A, 2, sum)      #col.sum = vector con las sumas de las columnas
cat("sumas de las filas = ", col.suma, " sumas de las columnas = " ,filas.suma)
# sumas de las filas =  6 15 24  sumas de las columnas =  12 15 18
```

Ejemplo 1.8 (Promedios)

Supongamos que tenemos una tabla con las notas de tres parciales (el porcentaje se indica en la tabla) y cuatro quices. El promedio de quices es un 25% de la nota.

Nombre	P1, 20 %	P2, 25 %	P3, 30 %	Q1	Q2	Q3	Q4
A	80	40	70	30	90	67	90
B	40	40	30	90	100	67	90
C	100	100	100	100	70	76	95

Para calcular los promedios podemos aplicar una función **FUN** que calcule el promedio sobre las filas de una matriz **notas**.

```
notas = matrix(c(80, 40, 70, 30, 90, 67, 90,
                40, 40, 30, 90, 100, 67, 90,
                100,100,100, 100, 70, 76, 95), nrow=3, byrow=TRUE); notas

# --- Cálculo de promedios. En la función, x representa cada fila
apply(notas, 1, function(x) 20/100*x[1]+ 25/100*x[2]+30/100*x[3]+ sum(25/400*x[4:7]) )
#[1] 64.3125 48.6875 96.3125
```

Para agregar la columna “promedios” a la matriz podemos usar **cbind()** = (concatenar columna),

```
proms=apply(notas, 1, function(x) 20/100*x[1]+ 25/100*x[2]+30/100*x[3]+ sum(25/400*x
[4:7]) )
cbind(notas, proms)

#
#                               proms
#[1,]  80  40  70  30  90 67 90 64.3125
#[2,]  40  40  30  90 100 67 90 48.6875
#[3,] 100 100 100 100  70 76 95 96.3125
```

Ejemplo 1.9 (Pesos baricéntricos)

El k -ésimo peso baricéntrico se define como $w_k = \frac{1}{(x_k - x_0) \cdots (x_k - x_{k-1}) (x_k - x_{k+1}) \cdots (x_k - x_n)}$

Para implementar los pesos baricéntricos de manera vectorial, observe que si $x = c(x_0, x_1, \dots, x_n)$ es un vector de datos y $n = length(x)$, entonces

```
X = matrix(rep(x, times=n), n, n, byrow=T)
```

Si $n = 4$, X es la matriz,

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_0 & x_1 & x_2 & x_3 \\ x_0 & x_1 & x_2 & x_3 \\ x_0 & x_1 & x_2 & x_3 \end{pmatrix}$$

entonces,

$$X - X^T = \begin{pmatrix} 0 & x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ x_0 - x_1 & 0 & x_2 - x_1 & x_3 - x_1 \\ x_0 - x_2 & x_1 - x_2 & 0 & x_3 - x_2 \\ x_0 - x_3 & x_1 - x_3 & x_2 - x_3 & 0 \end{pmatrix}$$

y los factores de los pesos baricéntricos están en las columnas de la matriz

$$X - X^T - I_{4 \times 4} = \begin{pmatrix} 1 & x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ x_0 - x_1 & 1 & x_2 - x_1 & x_3 - x_1 \\ x_0 - x_2 & x_1 - x_2 & 1 & x_3 - x_2 \\ x_0 - x_3 & x_1 - x_3 & x_2 - x_3 & 1 \end{pmatrix}$$

Generalizando, la columna k de $X - X^T + I$ tiene los factores $(x_k - x_i)$ que aparecen en la fórmula de w_k , es decir, w_k es el “producto de las entradas de la columna” k de la matriz $X - X^T + I$.

En R podríamos hacer

```
X = matrix(rep(x, times=n), n, n, byrow=T)
mD = X - t(X); diag(mD)=1
```

De esta manera, el k -ésimo peso baricéntrico sería

```
wk = prod(mD[, k])
```

Para calcular todos los pesos baricéntricos usamos `1/apply(mD, 2, prod)`.

■ Código R 1.43: Usando apply() para calcular pesos baricéntricos

```
x = seq(0, 0.5, by = 0.1)
n = length(x)
X = matrix(rep(x, times=n), n, n, byrow=T)
mD = X - t(X); diag(mD)=1
# vector de pesos baricéntricos
w = 1 / apply(mD, 2, prod) #(w1, w2,...,wn)
cat("x: ", x, "\n w:", w)
x: 0 0.1 0.2 0.3 0.4 0.5
w: -833.3333 4166.667 -8333.333 8333.333 -4166.667 833.3333
```

Ejemplo 1.10

Sea $\mathbf{x} = (x_0, x_1, \dots, x_n)$. Considere el problema de implementar una función que calcule todos los productos

$$L(k, a, \mathbf{x}) = (a - x_0) \cdot (a - x_1) \cdots (a - x_{k-1}) \curvearrowright (a - x_{k+1}) \cdots (a - x_n) \quad \text{para } k = 0, 1, \dots, n$$

El k -ésimo producto se salta el factor $(a - x_k)$. Por ejemplo,

$$\frac{k}{0} \quad L(k, a, \mathbf{x}) \\ L(0, a, \mathbf{x}) = (a - x_1) \cdot (a - x_2) \cdot (a - x_3) \cdots (a - x_n)$$

$$1 \quad L(1, a, \mathbf{x}) = (a - x_0) \cdot (a - x_2) \cdot (a - x_3) \cdots (a - x_n)$$

$$2 \quad L(2, a, \mathbf{x}) = (a - x_0) \cdot (a - x_1) \cdot (a - x_3) \cdots (a - x_n)$$

Podemos seguir la idea del ejemplo 1.9. Dado un vector de datos \mathbf{x} , sea

`X = matrix(rep(x, times=n), n, n, byrow=T)` con `n = length(x)`.

Si $n = 4$, entonces

$$\mathbf{X} = \begin{pmatrix} a & x_1 & x_2 & x_3 \\ x_0 & a & x_2 & x_3 \\ x_0 & x_1 & a & x_3 \\ x_0 & x_1 & x_2 & a \end{pmatrix}$$

Entonces haciendo `mN = a - X; diag(mN) = 1`, tendríamos que

$$mN = \begin{pmatrix} 1 & a-x_1 & a-x_2 & a-x_3 \\ a-x_0 & 1 & a-x_2 & a-x_3 \\ a-x_0 & a-x_1 & 1 & a-x_3 \\ a-x_0 & a-x_1 & a-x_2 & 1 \end{pmatrix}$$

Es esta manera, en el caso de n datos, tendríamos

$$\text{prod}(mN[k, 1]) = (a - x_0) \cdot (a - x_1) \cdots (a - x_{k-1}) \cdot 1 \cdot (a - x_{k+1}) \cdots (a - x_n)$$

Ahora, para calcular todos los productos $(a - x_0) \cdot (a - x_1) \cdots (a - x_{k-1}) \cdot (a - x_{k+1}) \cdots (a - x_n)$ desde $k = 0$ hasta $k = n$, solo debemos aplicar la función `prod()` a todas las filas de la matriz `mN`, y eso lo podemos hacer con la función `apply()`.

```
NL = function(a,x){
  n = length(x)
  X = matrix(rep(x, times=n), n, n, byrow=T) # por filas
  mN = a - X; diag(mN) = 1
  apply(mN, 1, prod)
}

# ---
x = seq(1, 5, by = 1)
NL(0.5, x)
# [1] 59.0625 19.6875 11.8125 8.4375 6.5625
```

1.10 Condicionales y Ciclos

■ **If.** La sintaxis es como sigue:

```
if(condición 1) {
resultado 1
}
```

```
if(condición 1) {
    result 1
} else {
    resultado 2
}
```

```
if(condición 1) {
    result 1
} else if (condición 2) {
    resultado 2
} else {
    resultado 3
}
```

Ejemplo 1.11 (Raíz n -ésima)

La raíz cúbica de un número real $x^{1/3}$ tiene dos soluciones complejas y una real. Nos interesa la solución real. Es usual en los lenguajes de programación que no esté definida la raíz cúbica (real) de un número. En **R** se obtiene $8^{(1/3)}=2$ pero $(-8)^{(1/3)}= \text{NaN}$. Una solución es separar el signo,

$$\sqrt[3]{x} = \text{sign}(x) \sqrt[3]{|x|}$$

■ Código R 1.44: Raíz cúbica

```
cubica = function(x) sign(x)*abs(x) $^{(1/3)}$ 

# pruebas-----
cubica(c(8, -8, -27, 27))
# 2 -2 -3 3
```

La raíz n -ésima (real) depende de si n es par o impar. La fórmula **sign**(x)***abs**(x) $^{(1/n)}$ sirve para el caso par o el caso impar, excepto en el caso que n sea par y $x < 0$, es decir, la fórmula sirve si n es impar o $x \geq 0$, en otro caso nos queda una expresión indefinida.

■ Código R 1.45: Raíz n -ésima

```
raiz = function(n,x){
  if(n%2 == 1 || x >=0 ){sign(x)*abs(x) $^{(1/n)}$ } else{ NaN }
}

# pruebas-----
c(raiz(3, -8), raiz(4, -64))
# [1] -2 NaN
```

La función `raiz(n, x)` del ejemplo anterior no está implementada para aplicarla a un vector `x`.

■ **Condicionales y vectores.** `ifelse()` aplica un condicional a vectores. Si necesitamos usar las conectivas **AND** u **OR**, es preferible usar la “forma corta”: `&` o `|` respectivamente, porque esta forma de ambas conectivas *permite recorrer vectores*. La sintaxis es

```
ifelse(condición sobre vector, salida 1, sino salida 2)
```

■ Código R 1.46:

```
a <- c(1, 1, 0, 1)
b <- c(2, 1, 0, 1)
# --- compara elementos de a y b
ifelse(a == 1 & b == 1, "Si", "No")
# [1] "No" "Yes" "No" "Yes"

# Observe que ifelse(a == 1 && b == 1, "Si", "No") retorna "NO" solamente.
```

Ahora podemos implementar la raíz n -énesima de manera que se pueda aplicar a vectores,

■ Código R 1.47: Raíz n -ésima

```
raiz = function(n,x){
  ifelse(n%%2==1 | x>0, sign(x)*abs(x)^(1/n), NaN) #else: n par y x negativo
}
# pruebas-----
raiz(3, c(8, -8, -27, 27))
# [1] 2 -2 -3 3
raiz(4, c(-64, 64, -32, 32))
#[1]     NaN   2.828427  NaN 2.378414
```

■ **Ciclo for.** La sintaxis es como sigue:

■ Código R 1.48:

```
for (contador-vector)
{
  Instrucción
}
```

■ Código R 1.49: Sumar con un ciclo “for”

```
# sumar de 1 a n
```

```
n = 5
suma =0
for(i in 1:n){ suma = suma + i
    print(suma)
}
#[1] 1
#[1] 3
#[1] 6
#[1] 10
#[1] 15
```

■ **Detener un ciclo.** Se usa la intrucción **break** para detener el ciclo (actual). También se puede usar **stop(...)** y lanzar un mensaje de error.

■ **Código R 1.50: "break" y "stop()"**

```
for (i in 1:10){
  if (i == 4) break
  print(i) # se detiene en i=4, así que solo imprime hasta 3
}
# Consola -----
[1] 1
[1] 2
[1] 3

# --- Usar stop() para indicar un error
if (f(a)*f(b)>0) stop("Se requiere cambio de signo") #Mensaje de error
```

■ **while.** La sintaxis es

■ **Código R 1.51: while**

```
while (condición)
{
  cuerpo
}
```

Por ejemplo,

■ **Código R 1.52:**

```
x <- 1
```

```
while (x <= 5)
{
    print(x)
    x <- x + 1 # esta es la manera de incrementar en R (no hay x++)
}

#---
#[1] 1
#[1] 2
#[1] 3
#[1] 4
#[1] 5
```

■ **repeat.** La sintaxis es

■ **Código R 1.53:**

```
repeat{
  ...
instrucciones
...
# until -- criterio de parada
if (condition) break
}
```

Por ejemplo,

■ **Código R 1.54:**

```
sum = 0
repeat{
  sum = sum + 2;
  cat(sum);
  if (sum > 11) break;
}
# Consola -----
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
```

Ejercicio 1.2 (*) La Criba de Eratóstenes se usa para generar listas de números primos menores que un número n dado. La descripción del método y un algoritmo lo puede encontrar en el libro [16], en las páginas 21 – 24. Implementar una función `cribaEratostenes(n)` que devuelve una lista con los primos $\leq n$. ■

1.11 La eficiencia de la vectorización.

Por supuesto, los ciclos `for` son a veces inevitables, pero hay una diferencia importante en el tiempo de corrida si vectorizamos la función. Para medir el tiempo de corrida usamos la función `system.time()` esta función retorna tres cosas: “user CPU time”, “system CPU time” y “elapsed time”.

El “user CPU time” es la cantidad de segundos que el CPU gastó haciendo los cálculos. El “system CPU time” es la cantidad de tiempo que el sistema operativo gastó respondiendo a las peticiones del programa. El “elapsed time” es la suma de los dos tiempos anteriores más tiempos de espera en la corrida del programa.

■ Código R 1.55: Sumas parciales – comparación

```
sumasparciales1 = function(f, a,k){ suma=0
                                for(i in a:k) suma = suma+f(i)
                                return(suma)
}
# Versión vectorizada
sumasparciales2 = function(f, a,k){ i = a:k
                                sum(f(i))
}

ui = function(i) 1/i^2
system.time(for(k in seq(100000,1000000, by = 100000)) cat( sumasparciales1(ui, 1, k)))
# user      system    elapsed
 3.852    0.000    3.854
system.time(for(k in seq(100000,1000000, by = 100000)) cat( sumasparciales2(ui, 1, k)))
#user      system    elapsed
 0.056    0.000    0.056
```

Si no podemos vectorizar, todavía podemos usar la librería `compiler` para acelerar el código.

■ Código R 1.56: Acelerar el código

```
k=100000000 # 100 millones
system.time(sumasparciales1(ui, 1, k))
# user      system    elapsed
# 72.240   0.032   72.331
system.time(sumasparciales2(ui, 1, k))
# user      system    elapsed
# 0.884   0.348   1.233
```

```
# ---Acelerar el código
library(compiler)
cf1 = cmpfun(sumasparciales1) # cf1 es la nueva función con el cuerpo
# de sumasparciales1() compilado
# Ahora usa cf1(f,a,k)

system.time(cf1(ui, 1, k))
# user     system   elapsed
# 56.740   0.040   56.831
```

La función **cmpfun()** compila el cuerpo de la función y devuelve una nueva función con su cuerpo “compilado”. Como se ve en el ejemplo anterior, al aplicar la función **cmpfun()** a la función **sumasparciales1**, pasamos de **72.240** segundos a **56.740** al evaluar la suma de la serie con **k=100000000**. Aún así no es tan bueno comparado con el desempeño de la función vectorizada **sumasparciales2** que solo ocupó **0.884** segundos para hacer la misma suma.

La función **apply()** en general no es más rápido que un ciclo **for** porque esta función **apply** crea su propio ciclo. Pero en **R** hay otras funciones dedicadas para operaciones con filas y columnas que si son mucho más veloces. En todo caso, cuando un programa presenta muchos ciclos (posiblemente anidados) y si se requiere eficiencia, es probable que lo mejor sea usar **R** en la parte en que **R** es fuerte (manipulación de objetos) y crear código **C++** compilado para esos ciclos y llamar las funciones respectivas desde **R** (o usar paquetes como RcppArmadillo para “acelerar **R** con el alto rendimiento de C++ en álgebra lineal”). Este es un tópico que no se toca en este libro.

1.12 Funciones II

Recordemos que las funciones se declaran usando la directiva **function()** y es almacenada como un objeto. Las funciones tienen argumentos y estos argumentos podrían tener valores por defecto. La sintaxis sería algo como

```
nomrefun <- function(a1,a2,...,an) {
## cuerpo
instrucc-1
instrucc-2
...
salida # o también return(salida)
}
```

Ejemplo 1.12 (Una función a trozos).

Implementar la función $f(x) = \begin{cases} \cos x & \text{si } x < 0 \\ \operatorname{sen} x & \text{si } x \geq 0 \end{cases}$

Solución:

■ Código R 1.57:

```
f = function(x){if(x<0) cos(x) else sin(x)}
f(pi)
#[1] 1.224647e-16
f(pi/2)
#[1] 1
```

■ Salida con listas. Se puede usar listas (`list()`) para acceder a los componentes de la salida por nombre usando \$

■ Código R 1.58: Operador “\$”

```
fun2xy <- function(x, y) {
  s <- x + y
  r <- x - y
  list(lasuma = s, laresta = r) # valor a retornar
}
# pruebas -----
fun2xy(5,4)
$lasuma
#[1] 9
$laresta
#[1] 1
fun2xy(5,4)$lasuma
#[1] 9
fun2xy(5,4)$laresta
#[1] 1
```

■ Variables locales. Las variables declaradas en el cuerpo de la función son variables locales de la función.

■ Código R 1.59: Variables locales

```
funxy <- function(x, y) {
  s <- x + y      # s es variable local
```

```

r <- x - y      # r es variable local
c(s, r)         # retorna un vector con dos valores
}

funxy(5,4)
# [1] 9 1
funxy(5,4)[1]
# [1] 9
funxy(5,4)[2]
# [1] 1
funxy(c(2,2,2), c(3,3,3)) # funxy también recibe vectores o matrices
# [1] 5 5 5 -1 -1 -1

```

■ **Argumentos por defecto.** Se puede declarar argumentos por defecto en la declaración de argumentos escribiendo `function(arg1=valor1, arg2, arg3, arg4=valor4,...)`,

■ Código R 1.60: Argumentos defecto

```

fun3xy = function(x=2,y=3){ x+y }

fun3xy()          # = ffun3xy(2,3)
#[1] 5
fun3xy(5,9)
#[1] 14
fun3xy(10, )     # = ffun3xy(10,3)
#[1] 13
fun3xy( ,8)      # = ffun3xy(2,8)
#[1] 10
>

```

■ **Usando funciones anónimas.** A veces podemos usar una función `function(){...}` sin declarar su nombre.

■ Código R 1.61: Función anónima

```

# Declarar una función anónima y la evaluar en 1.1
(function(x) sin(x))(1.1)
#[1] 0.8912074

# El código anterior corresponde a
f <- function(x) sin(x)
f(1.1)
#[1] 0.8912074

```

Por ejemplo, la función **integrate(f, a, b)** aproxima $\int_a^b f(x) dx$. La función f la podemos incluir en formato anónimo: Digamos que queremos aproximar $\int_0^1 e^{-x^2} dx$,

■ **Código R 1.62:**

```
integrate(function(x) exp(-x^2), 0, 1)
# 0.746824132812 with absolute error < 0.0000000000000083
```

Otro ejemplo, el comando **outer** requiere una función **FUN** que puede ser declarada de manera anónima,

■ **Código R 1.63:**

```
misuma = function(n){
  y = seq(n, 1, by = -1)
  x = 1 : n
  x = outer(y,x,FUN = function(i,j) i/j^2) # función anónima, x = (5,4/2^2,3/3^2,..., 1/n^2)
  sum(x)
}
misuma(5)
#[1] 21.95417
```

Podemos aplicar la función **sapply(x, FUN)** con funciones anónimas,

■ **Código R 1.64:**

```
fc= function(x) sapply(x, function(x) x^2)
fc(c(1,3,5,6))
#[1] 1 9 25 36

#Función de varios argumentos
fstx = function(s,t,x) s+t+x
f = function(x) sapply(x, function(x) fstx(1,2,x))
f(c(1,3,5,6))
#[1] 4 6 8 9
```

1.13 Expresiones, tiras y funciones

Las expresiones son hileras de caracteres que tienen sentido para **R**, por ejemplo los comandos base de **R**: **outer**, **mean**, etc. Los comandos que escribimos *son evaluados* por **R** y se ejecutan si son válidos (evaluados e incorporados a la sesión – no debe confundirse con “evaluación numérica”).

Algunas funciones que vamos a utilizar, reciben como argumento una expresión. El problema aparece cuando la función que definimos recibe como argumentos tiras (strings), expresiones o funciones por nombre.

Las *expresiones* se convierten en funciones usando **eval()**

■ Código R 1.65: Convertir expresión en función

```
# --- Expresión
fxx = expression(x^2+1/x)
# --- convertir la expresión en una función
f = function(x){eval(fxx)}
# --- Evaluar numéricamente
f(3)
# [1] 9.333333
```

Las tiras (strings) se convierten en expresiones usando **parse()**. Esta función devuelve una expresión.

■ Código R 1.66: Convertir una tira (string) en una función

```
# --- Tira
fs = "sqrt(x)"
fxpr = parse(text = fs) # devuelve expression(sqrt(x))
# --- convertir en una función
f = function(x){eval(fxpr)}
# --- Evaluar numéricamente
f(4)
# [1] 2
```

■ Componentes de una función. Las funciones en R tienen tres componentes

- body()**: el código dentro de la función
- formals()**: la lista de argumentos que controlan las llamadas a la función
- environment()**: el “mapa” de la localización de las variables de las funciones. variables.

■ Código R 1.67: Componentes de una función

```
fa = function(x) sqrt(x)
fb = function(x){ sqrt(x) }
# --- formals
formals(fa)
# $x
formals(fb)
# $x
```

```
# --- cuerpo
body(fa)
# sqrt(x)
body(fb)
# {
#   sqrt(x)
# }
# --- environment
environment(fa)
# <environment: R_GlobalEnv>
```

Se puede acceder al contenido de la función **f** con los componentes de la lista **body(f)**,

■ Código R 1.68: body(f)

```
fa = function(x) sqrt(x)
fb = function(x){ sqrt(x) }

body(fa)[[1]]
body(fa)[[2]]
body(fb)[[1]]
body(fb)[[2]]

# Consola -----
> body(fa)[[1]]
sqrt
> body(fa)[[2]]
x
> body(fb)[[1]]
{
> body(fb)[[2]]
sqrt(x)
>
```

■ **Pasar funciones como argumentos de una función.** En R podemos pasar una función como argumento a otra función de varias maneras.

- Una función puede pasar “por nombre”

■ Código R 1.69: función con una función como argumento

```
fx = function(f,x){
  f(x)
```

```

        }
f = function(x) sqrt(x)
fx(f,4)
# [1] 2

```

- Si la función pasa como una tira, entonces usamos **parse()** para convertirla en expresión.

■ Código R 1.70: función con una “tira” como argumento

```

funstring = function(f,a){ #f es tira (string)
  g = parse(text = f)
  fx = function(x) eval(g) # o también eval(g[[1]])
  fx(a)
}
funstring("sqrt(x)", 4)
# 2

```

- Si la función pasa como una expresión, entonces en caso necesario, usamos **eval()** para convertirla en función.

■ Código R 1.71: función con una expresión como argumento

```

fexpr = function(f,a){ #f es expresión
  fx = function(x) eval(f)
  fx(a)
}
fexpr(expression(sqrt(x)), 4)
# 2

```

■ Derivación simbólica. Para algunos algoritmos necesitamos calcular derivadas simbólicamente, en la medida que se pueda. La función “derivada”: **D()** *recibe expresiones y devuelve expresiones*. La función **D()** reconoce las operaciones **+**, **-**, *****, **/** y **^**, y las funciones **exp**, **log**, **sin**, **cos**, **tan**, **sinh**, **cosh**, **sqrt**, **pnorm**, **dnorm**, **asin**, **acos**, **atan**, **gamma**, **lgamma**, **digamma**, **trigamma** y **psigamma** (esta última solo respecto al primer argumento).

En el siguiente código vamos a ver como pasar una función por nombre y una función como tira, e internamente convertirla en expresión y que pueda ser usada por **D()**.

Para pasar la función como una tira, solo hay que recordar que la función **parse()** convierte la tira en expresión, que es lo que necesita recibir **D()**.

■ Código R 1.72: Pasar una función a “D()” como tira

```

fun2der = function(f, a){ # f is string
  g = parse(text=f)
  # convertir en función
  fx = function(x){eval(g[[1]])}
  # calcular la derivada
  g. = D(g,"x")
  # convertir la derivada en función
  fp = function(x){eval(g.)}
  cat(fx(a))
  cat(fp(a))
}

fun2der('sqrt(x)', 4)
# [1] 2
# [1] 0.25

```

Si la función la llamamos “por nombre”, entonces debemos tomar en cuenta si la función fue definida usando llaves o no, porque debemos tomar el cuerpo de la función (**body**). Para crear un estándar, usamos la función **call** para agregar las llaves en caso de que el cuerpo de la función no las tenga.

■ Código R 1.73: Pasar una función a “D()” por nombre

```

fa = function(x){sqrt(x)} # con llaves
fb = function(x) sqrt(x) # sin llaves

lafun = function(f, a){
  # Si no tiene llaves, agregar las llaves
  bf = body(f)
  if(bf[[1]]!="{" ) bf = call("{", bf)
  #la función tiene llaves
  g. = D(bf[[2]], "x")
  fp = function(x){eval(g.)}
  fp(a)
}
lafun(fa, 4) # fp = 1/(2*sqrt(4))
lafun(fb, 4)

# [1] 0.25
# [1] 0.25

```

■ **Derivadas de orden superior.** En la “ayuda” de **R** aparece el código de una función para calcular derivadas de orden superior: **DD(expresion(f), "var", n)**

■ Código R 1.74: Función derivada n -ésima “DD()”

```
# DD Calcula derivadas de orden superior, recibe una expresion--
DD <- function(expr, name, order = 1) {
  if(order < 1) stop("order must be >= 1")
  if(order == 1) D(expr, name)
  else DD(D(expr, name), name, order - 1)
}

DD(expression(x^5), "x", 3)

# 5 * (4 * (3 * x^2))
```

Ejemplo 1.13 (Polinomio de Taylor).

El polinomio de Taylor de orden n de f alrededor de $x = a$ es

$$P_n(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \dots + \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Se tiene que

$$f(x) = P_n(x) + R_n(f) \text{ o también } f(a+h) \approx P_n(a) \text{ si } h \text{ es pequeño.}$$

■ Código R 1.75: Polinomio de Taylor de orden n , alrededor de $x = a$

```
taylorT = function(f, x0, a, n){ # f es tira
  # parse devuelve una expresión
  g = parse(text=f)
  # convertir en función
  fx = function(x){eval(g[[1]])}
  # almacenar los sumandos
  smds = rep(NA, length=n+1)
  for(k in 1:n){
    g. = DD(g,"x", k)
    fp = function(x) eval(g.)
    smds[k]=1/factorial(k)*(x0-a)^k *fp(a)
  }
  smds[n+1] = fx(a)
  sum(smds)
}
taylorT("sin(x)", 1.1, 1, 2)
# [1] 0.8912939
```

■ **Derivación numérica.** Hay funciones que no podemos derivar de manera simbólica, por la manera en que han sido definidas (por integrales, por ecuaciones diferenciales, etc.). Para efectos de hacer aproximaciones, en general no se requiere la función derivada sino una aproximación bastante buena. En la sección 7.1 veremos que podemos cambiar la función **DD()** por aproximaciones basadas en “diferencias finitas” (al menos para **n = 1:4**).

Ejercicio 1.3 ■ Implementar **taylorN(f, x0, a, n)** donde **f** viene dada por nombre. Por ejemplo

```
taylorN = function(fun,x0, a, n){ #fun es el nombre de la función
  ...
}

# --- Pruebas
# definirde la función por nombre
fun = function(x) sqrt(x)

# taylorN llama a fun por nombre
taylorN(fun,x0, a, n)
```

Usar la implementación anterior para

- Aproximar $\sin(1.1)$ con un polinomio de Taylor de orden $n = 2$ y otro de orden $n = 12$, ambos alrededor de $a = 1$.
- Aproximar $\sqrt{0.0088}$ con un polinomio de Taylor de orden $n = 2$ y otro de orden $n = 12$, ambos alrededor de $a = 0.05$. Observe el comportamiento del polinomio de orden 12 (La función es $f(x) = \sqrt{x}$).

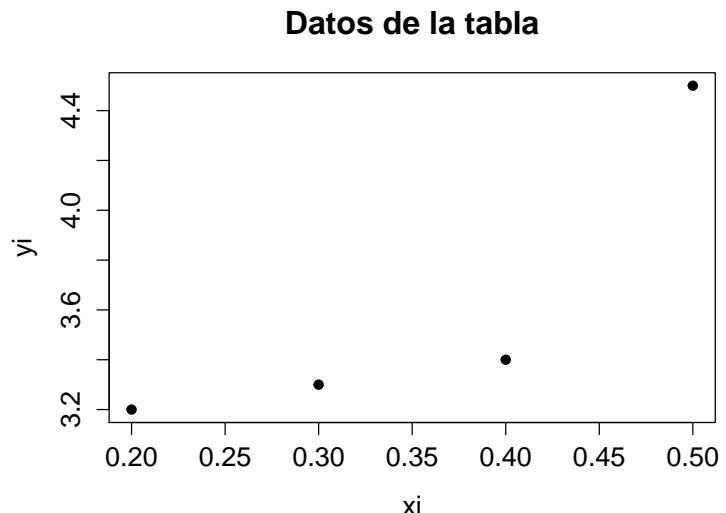
1.14 Gráficos en R

Gráficas de dispersión (scatter plot).

Supongamos que tenemos un colección de datos, como los de la tabla 1.1, podemos hacer una representación gráfica de estos datos **x, y** usando **plot(x, y, opciones)**.

La representación gráfica que se obtiene con **R** se muestra a la derecha (el gráfico se guarda en formato .pdf en la cejilla **Export**).

<i>x</i>	<i>y_i</i>
0.2	3.2
0.3	3.3
0.4	3.4
0.5	4.5

Tabla 1.1: Datos (*x_i, y_i*)

El código que se usó fue

■ **Código R 1.76: "plot()"**

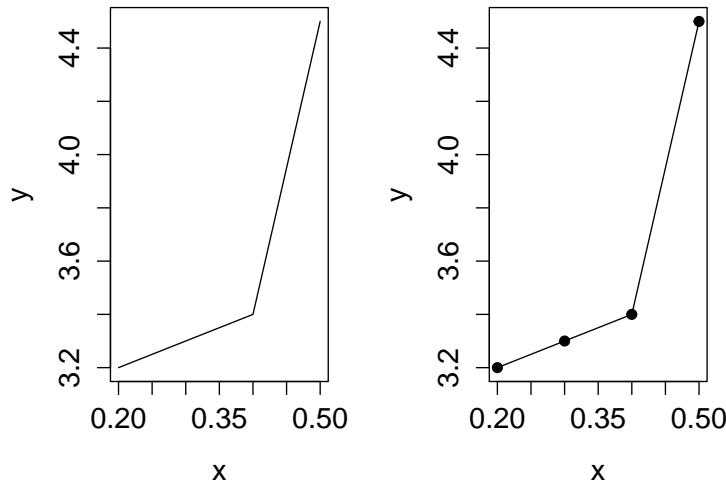
```
x = seq(0.2, 0.5, by=0.1)
y = c(3.2, 3.3, 3.4, 4.5)
plot(x,y,
      pch = 19, # símbolo para los puntos
      main = "Datos de la tabla",
      xlab = "xi",
      ylab = "yi"
    )
```

También se pueden usar líneas continuas con la opción **type = "l"**

```
plot(x,y, type = "l")

plot(x,y, type = "o", pch = 19)

par(mfrow=c(1,2)) # Agrupar los gráficos en una fila, dos columnas
```



■ **La función `curve()`.** En general, se puede usar el comando `curve(f, a, b)` para hacer la representación gráfica de una función $y = f(x)$ con $x \in [a, b]$ (con un solo argumento). La variable por defecto es `x`. El nombre de la variable se puede cambiar, si fuera necesario, con el parámetro `xname`. Por ejemplo

■ Código R 1.77: Usando `curve()`

```
# Graficar cos(x)
curve(cos, -pi, 3*pi, n = 1001, col = "blue") # n = número de puntos

# Graficar f(t)
ft = function(t) sin(t)
curve(ft, -pi, pi, xname = "t")

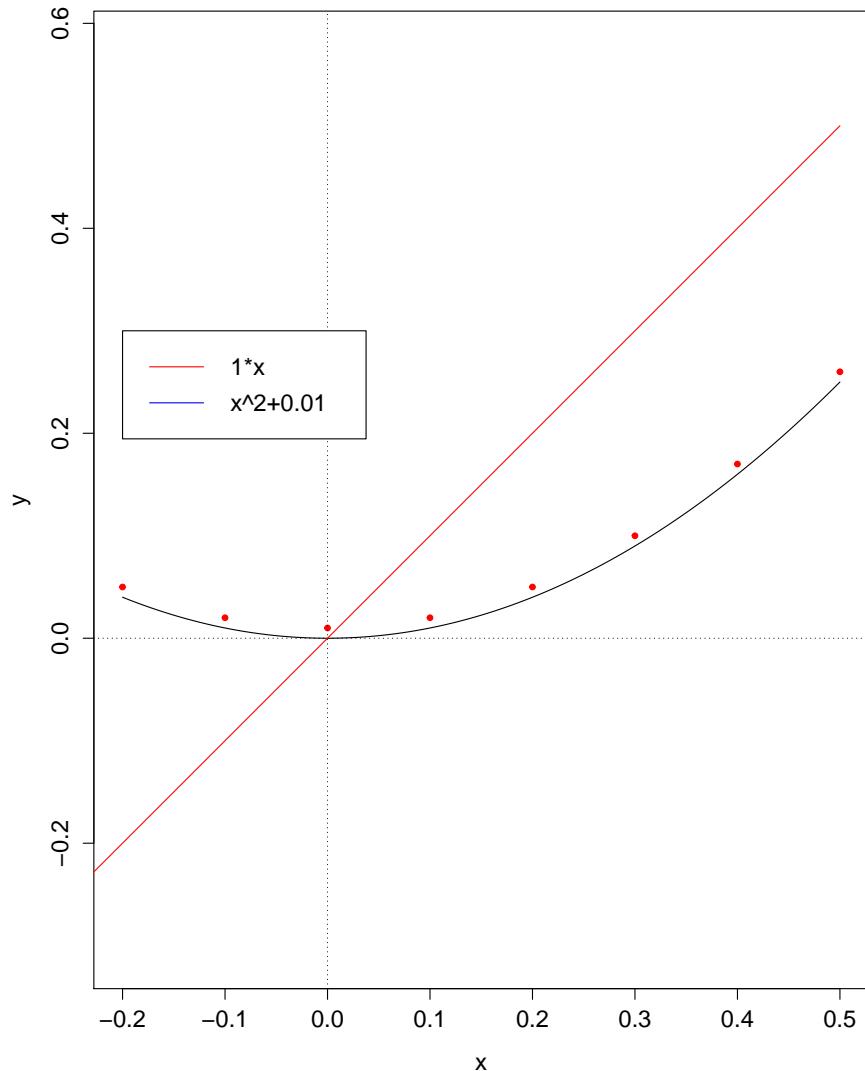
# Graficar la derivada de f
f      = expression(x^2+3*x-3)
df     = D(f, 'x')
fdf   = function(x) eval(de)
curve(fdf, 0, 10)
```

■ **Datos y curvas.** Para graficar datos y curvas en un mismo sistema, primero se usa `plot(x,y)` para los datos y luego `curve(f, a, b, add=T)` para la función. `add=T` se usa para agregarla al gráfico anterior (ya existente).

■ Código R 1.78: “`plot`” y “`curve`”

```
x = seq(-0.2, 0.5, by=0.1)
y = x^2+0.01
plot(x,y, pch = 19, cex=0.7, col= "red", asp=1) # asp = 1: Usar la misma escala en ambos ejes
curve(x^2+0.01, -0.2, 0.5, add=T)
```

```
# Ejes X y Y con líneas punteadas
abline(h=0,v=0, lty=3)
curve(x^2, -0.2, 0.5, add=T)
curve(1*x, -1, 0.5, col= "red", , add=T) # notar 1*x
legend(-0.2,0.3, c("1*x", "x^2+0.01"), col = c("red","blue"), lty=1)
```



■ **Gráficas de funciones que reciben vectores.** Consideremos la función

$$L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1}) \curvearrowright (x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1}) \curvearrowright (x_k - x_{k+1}) \cdots (x_k - x_n)} \text{ para } k = 0, 1, 2, \dots, n$$

Siguiendo la construcción que hicimos en el ejemplo 1.9, podemos implementar una función **Ln1**, para el caso $k = 1$ de la siguiente manera,

```
Ln1 = function(x){
  k = 1
  vxsk = xi[-(k+1)] # x sin entrada k+1
  salida = prod((x - vxsk)/(xi[k+1]-vxsk))
  return(salida)
}
```

Un primer intento para hacer la representación gráfica de esta función nos da varios errores,

```
xi = seq(-1, 5, by =1)
curve(Ln1, -1 , 5, col = 2, n = 100); abline(h=0,v=0, lty=3)
# Error en curve(Ln1, -1, 5, col = 2, n = 100) :
# 'expr' did not evaluate to an object of length 'n'
#...
```

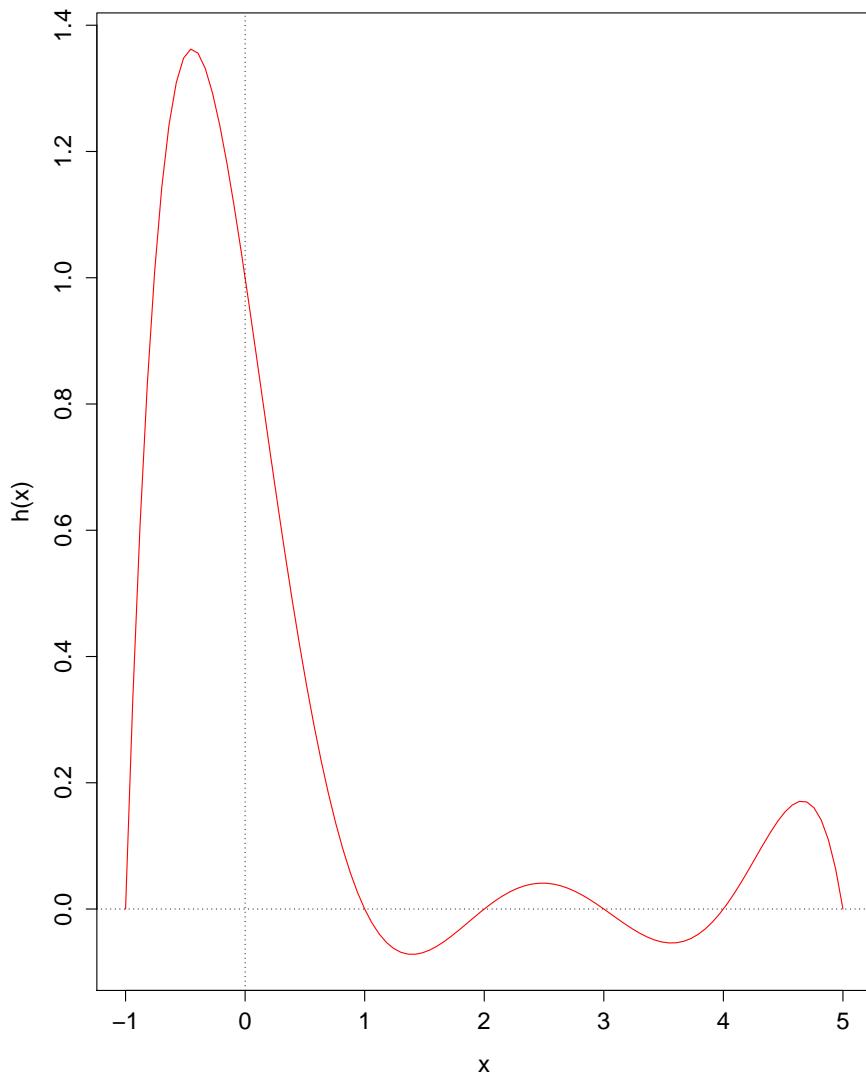
Ln1 debe ser vectorizada: Esto significa que, si se evalúa un vector, tiene que devolver un vector de la misma longitud (no un escalar). **curve** requiere que esta función use un solo argumento y que este vectorizada.

Para resolver el problema se puede usar la función **Vectorize()**,

■ Código R 1.79: Usando Vectorize() para vectorizar

```
Ln1 = function(x){
  k = 1
  vxsk = xi[-(k+1)] # x sin entrada k+1
  salida = prod((x - vxsk)/(xi[k+1]-vxsk))
  return(salida)
}

h = Vectorize(Ln1) # vectorizar Ln1
xi = seq(-1, 5, by =1)
curve(h, -1 , 5, col = 2, n = 100); abline(h=0,v=0, lty=3)
```



■ **Grafica de funciones con varios argumentos.** Pudimos implementar la función `Lnk` de una manera más natural como `Lnk(xi,k,x)`,

■ Código R 1.80:

```
Lnk = function(xdat, k, x){
  vxsk = xdat[-(k+1)] # x sin entrada k+1
  salida = prod((x - vxsk)/(xdat[k+1]-vxsk))
  return(salida)
}
```

Para hacer la representación gráfica de la función `Lnk(xi,1,x)` usando `curve()`, podemos usar la función `sapply()`

porque necesitamos hacer que esta función aparezca como función de un solo argumento, **x** en este caso, y además que sea vectorizada.

Recordemos que la sintaxis breve de esta función es **sapply(x, FUN)** es decir, aplica la función **FUN** al vector **x** y *devuelve un vector de la misma longitud*. En el caso de **curve()**, el vector **x** es el vector que usa **curve** para evaluar la función.

■ Código R 1.81: Usando **sapply()** para vectorizar

```
Lnk = function(xdat, k, x){
  vxsk = xdat[-(k+1)] # x sin entrada k+1
  salida = prod((x - vxsk)/(xdat[k+1]-vxsk))
  return(salida)
}
xi = seq(-1, 5, by =1)
curve(sapply(x, function(x) Lnk(xi,1,x) ), -1, 5); abline(h=0,v=0, lty=3)
```

Ejercicio 1.4 Sea

$x = (0.41040018, 0.91061564, -0.61106896, 0.39736684, 0.37997637, 0.34565436, 0.01906680, -0.28765977)$

Considere la función $\ell(\alpha) = \sum_{i=1}^n \frac{x_i}{1 + \alpha x_i}$ con **n** = **length(x)**. Implemente la función $\ell(\alpha)$ y haga la representación gráfica con $-1 \leq \alpha \leq 1$.

■

1.15 Correr un programa (script)

Supongamos que hemos implementado un programa llamado **miprograma.r**. Un programa (o un script) solamente es un conjunto de instrucciones: Reciben una entrada, hace cálculos y tiene una salida. Hay dos maneras de correr el script: Abrir el archivo (digamos en RStudio) y seleccionar y correr el código (también se puede usar el botón **→ Source**) o usar el comando **source(miprograma.r)**

Podemos llamar un script desde un directorio específico. Por ejemplo si el script está en el directorio **/home/walter-15/Escritorio/RScripts** (usando Ubuntu) entonces lo llamaríamos con

```
source从根本路径到miprograma.r)
```

En Windows sería algo como

```
source从根本路径到miprograma.r)
```

Como los programas en **R** están en un ambiente en el que posiblemente se han definido otras variables, una buena costumbre es limpiar el espacio de trabajo antes de correr nuestro programa. Así que todo programa debería iniciar

con la instrucción

```
rm(list=ls())
```

Como un ejemplo, implementamos un programa simple que calcula las raíces reales de una función cuadrática. Luego lo guardamos en algún directorio como **cuadratica.R**. En nuestro caso, lo guardamos en el directorio /home/walter-15/Escritorio/RScripts/.

■ Código R 1.82: script “cuadratica.R”

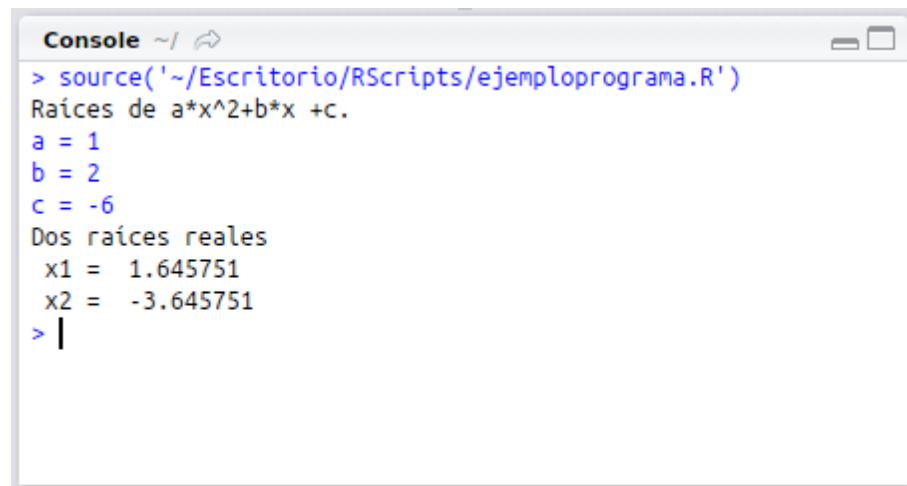
```
# Raíces de a*x^2+b*x +c
# --- Limpiar el ambiente de trabajo
rm(list=ls())
cat("Raíces de a*x^2+b*x +c.\n")
# Lectura de coeficientes desde el teclado (ENTER)
a = as.numeric(readline("a = ")) # Leer desde el teclado (en la consola)
b = as.numeric(readline("b = "))
c = as.numeric(readline("c = "))

# Cálculos
if(a==0) stop("No es cuadrática")
# Discriminante
d = b^2-4*a*c
# ---
if(d==0){
  x1 = -b/(2*a)
  cat("Una raíz real x1 = ", x1)
}
if(d>0){
  x1 = (-b+sqrt(d))/(2*a)
  x2 = (-b-sqrt(d))/(2*a)
  cat("Dos raíces reales \n", "x1 = ", x1, "\n", "x2 = ", x2)
}
if(d<0) cat("Las raíces son complejas")
```

Para correrlo, abrimos por ejemplo RStudio y ejecutamos la instrucción

```
source("/home/walter-15/Escritorio/RScripts/cuadratica.R")
```

Un ejemplo de corrida se ve en la figura que sigue,



```

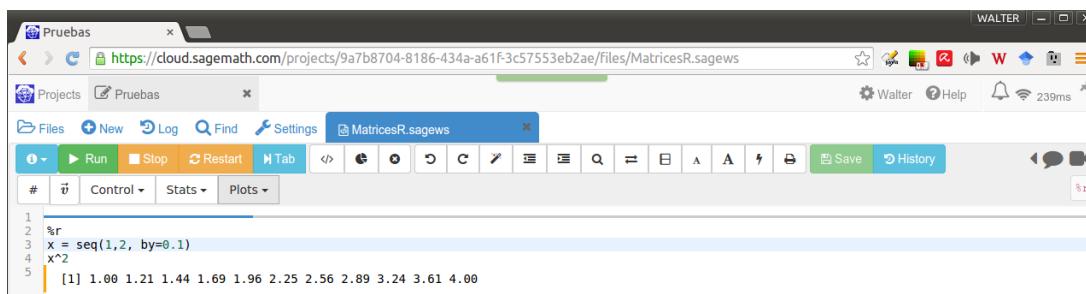
Console ~/ ↗
> source('~/Escritorio/RScripts/ejemploprograma.R')
Raices de a*x^2+b*x +c.
a = 1
b = 2
c = -6
Dos raices reales
x1 = 1.645751
x2 = -3.645751
> |

```

■ **Bibliotecas.** Por supuesto, el script también puede ser solamente una biblioteca de funciones personales que llamamos con la función **source()** al principio de otro script

1.16 R en la nube.

Hay varios sitios en Internet para usar R en la nube. Tal vez la mejor manera sea en la página de SAGEMATH (<http://www.sagemath.org/>). Después de registrarse, elige “Use SageMath Online”. La primera vez puede crear un proyecto y luego crea un archivo nuevo (“Create new files”) y después elige “SageMath Worksheet”. En la nueva página, en la pestaña “Modes” elige R. Una gran ventaja es que podrá también usar SAGEMATH.



The screenshot shows a web-based SageMath worksheet interface. The URL in the address bar is <https://cloud.sagemath.com/projects/9a7b8704-8186-434a-a61f-3c57553eb2ae/files/MatricesR.sagews>. The workspace contains the following R code:

```

1 %r
2 x = seq(1,2, by=0.1)
3 x^2

```

The output of the code is:

```

[1] 1.00 1.21 1.44 1.69 1.96 2.25 2.56 2.89 3.24 3.61 4.00

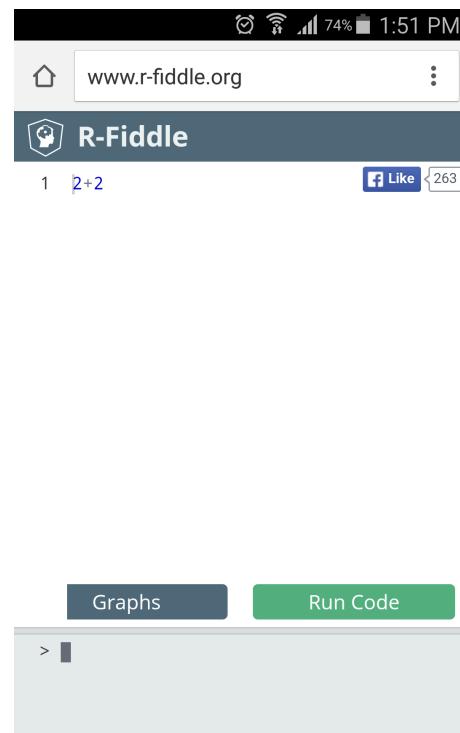
```

Figura 1.6: R en el entorno de SageMath en la nube

También se puede instalar la app “Sage Math” para Android desde Google Play.



Figura 1.7: SageMath en Android



■ **R en Chrome.** También se puede tener R en el celular si instalamos el navegador Chrome y vamos al sitio www.r-fiddle.org. En Firefox, a la fecha, funciona pero con algunos problemas en el teclado.



2 — Aritmética del Computador y Errores

2.1 Introducción

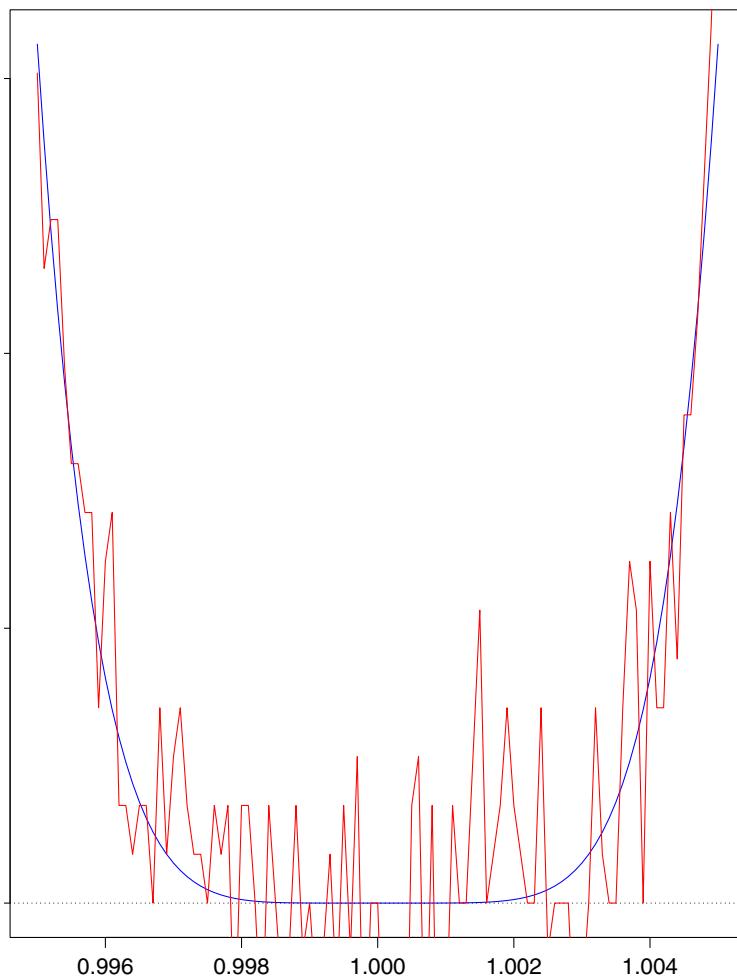
El computador usa una cantidad finita de dígitos para representar un número (a lo sumo 16 dígitos). Como los números están, en general, representados de manera inexacta entonces las operaciones aritméticas también se hacen de manera inexacta. El error relativo sin embargo es muy pequeño para una sola operación, el problema es que el error se podría acumular en dos o más operaciones.

Por ejemplo, considere las función $f(x) = (1 - x)^6$. Esta función, en esta forma, requiere pocos cálculos para ser evaluada; pero si la expandimos entonces $f(x) = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$, y ahora si podríamos tener una acumulación de error,

```
f = function(x) (1-x)^6
ef = function(x) x^6-6*x^5+15*x^4-20*x^3+15*x^2-6*x+1
options(scipen = 999)
x = seq(0.995, 1.005, by=0.0001)
y1 = f(x)
y2 = ef(x)
(y1-y2)
# [1] 0.0000000000005259668651  0.000000000000022949677449
# [3] -0.00000000000002039074118 -0.000000000000016552825468
# [5] -0.00000000000002956657207  0.00000000000003101598477
```

Probablemente un gráfico nos ayude a visualizar los errores de redondeo.

```
x = seq(0.995, 1.005, by=0.0001)
a = x[1]; b = x[length(x)]
curve((1-x)^6, a, b, col="blue"); abline(h=0,v=0, lty=3)
curve(x^6-6*x^5+15*x^4-20*x^3+15*x^2-6*x+1, a, b, col="red", add=T)
```



■ **Épsilon de la máquina.** El computador solo puede aproximar los números reales y hay un límite para esta aproximación, en particular, el “épsilon de la máquina” es el más pequeño número positivo que el computador puede representar. Este número `eps` $\neq 0$ es el más pequeño número tal que `eps+1!=1` o también `1-eps !=1`. En **R** podemos calcular las dos variantes,

```
str(.Machine[c("double.digits", "double.eps", "double.neg.eps")], digits=10)
# List of 3
# $ double.digits : int 53
# $ double.eps     : num 2.220446049e-16
```

```
# $ double.neg.eps: num 1.110223025e-16
```

Esto nos dice que los épsilon de la máquina son 2^{-52} y 2^{-53} . Ahora, haciendo un pequeño cálculo:

$$2^{-53} = 10^{-x} \implies x = 53 \cdot \log_1 0(2) \approx 15.95459,$$

es decir, el computador trabaja con a lo sumo, dieciséis dígitos en doble precisión.

- a.) El rango de enteros que pueden ser representados de manera exacta es $[2^{-53}, 2^{53}]$, es decir, el computador puede manejar enteros de manera exacta entre -10^{-17} y 10^{-17} más o menos.

```
.Machine$integer.max #El más grande entero que se puede representar en la máquina
```

```
# [1] 2147483647
```

- b.) El máximo valor que puede ser representado es

$$1.797693 \times 10^{308}$$

```
.Machine$double.xmax # El más grande real que se puede representar en la máquina
```

```
# [1] 1.797693e+308
10^309
# [1] Inf
```

- c.) El mínimo valor positivo que puede ser representado es

$$2.225074 \times 10^{-308}$$

- d.) Si $rd(x)$ es la representación de x (por redondeo simétrico) en doble precisión, entonces

$$\left| \frac{x - rd(x)}{x} \right| \leq 2^{-52} \leq 0.5 \times 10^{-15}$$

En R se puede usar la función `all.equal(x, y, tol)` que devuelve `TRUE` tan pronto como `x` y `y` están a una distancia $\leq tol$. El valor por defecto de `tol` es 10^{-8} .

■ **Errores de truncación y errores de redondeo.** Hay básicamente dos fuentes de error en los cálculos aproximados: Los errores de truncación y los errores de redondeo.

Los errores de truncación están relacionados con los modelos matemáticos que usamos para resolver un problema, por ejemplo al reemplazar una derivada por una diferencia finita o al “discretizar” un problema de naturaleza continua.

Los errores de redondeo aparecen por la necesidad de redondear números con expansión decimal infinita. En el computador puede pasar que $a + \epsilon = a$ aún cuando $\epsilon \neq 0$.

■ **Inestabilidad numérica y problemas mal condicionados.** La introducción de errores de redondeo podría afectar seriamente la precisión de una solución numérica (solución aproximada). La calidad de la solución podría no ser aceptable por dos condiciones:

- Los errores de redondeo son amplificados por el algoritmo (“inestabilidad numérica”)
- Pequeñas perturbaciones en los datos producen grandes cambios en la solución (“problema mal condicionado” o “problema sensitivo”)

Más adelante veremos ejemplos de estas situaciones.

■ **Decimales correctos y dígitos significativos en una aproximación.** Los cálculos numéricos son llevados a cabo con un número finito de dígitos aunque la mayoría de números tengan un número infinito de dígitos. Cuando los números son muy grandes o muy pequeños se representan con *punto flotante*,

$$x = p \times 10^q$$

donde p es un número cercano a 1 y q es un entero; por ejemplo $0.00147 = 0.147 \times 10^{-2}$.

Cuando contamos el número de dígitos en un valor numérico, las *cifras significativas* son los dígitos usados para expresar el número: 1, 2, 3, 4, 5, 6, 7, 8 y 9. El cero es también significativo, excepto cuando es usado para fijar el punto decimal o para llenar los lugares de dígitos desconocidos o descartados. Por ejemplo, 0.147 y 3.23 tienen tres cifras significativas mientras que 0.000207 = 0.207×10^{-3} tiene tres cifras significativas.

Sea \tilde{p} es una aproximación de p (digamos que con más de k decimales); decimos que \tilde{p} tiene k *decimales correctos*, si k es el natural más grande tal que $|\tilde{p} - p| \leq 0.5 \times 10^{-k}$.

Ejemplo 2.1

Si $p = 0.001234$ y $\tilde{p} = 0.001234 \pm 0.000004$, entonces $|\tilde{p} - p| \leq 0.5 \times 10^{-5}$ (pero $|\tilde{p} - p| > 0.5 \times 10^{-6}$) así que \tilde{p} tiene 5 decimales correctos. En efecto, $0.001234 + 0.000004 = 0.001238$ y $0.001234 - 0.000004 = 0.00123$

Si \tilde{p} tiene k decimales correctos, los *dígitos* en \tilde{p} que ocupan posiciones donde “la unidad” es $\geq 10^{-k}$ se llaman *dígitos significativos* (es decir, los ceros iniciales no cuentan).

Ejemplo 2.2

Por ejemplo, si $\tilde{p} = 0.001234$ aproxima p con 5 decimales correctos, entonces \tilde{p} tiene tres dígitos significativos pues $0.001, 0.0002, 0.00003$ son $> 10^{-5}$ pero $0.000004 > 10^{-5}$.

Ejemplo 2.3

Si sabemos que $\tilde{p} = 1.5756457865$ es una aproximación de un número desconocido p y que $|p - \tilde{p}| \leq 0.000000005$, entonces al menos sabemos que $|p - \tilde{p}| \leq 0.5 \times 10^{-7}$ (pero no sabemos si $|p - \tilde{p}| \leq 0.5 \times 10^{-8}$) con lo que \tilde{p} tiene *como mínimo*, 7 decimales correctos.

■ **Error absoluto y error realtivo.** Sea \tilde{p} es una aproximación de p , entonces,

- $|\tilde{p} - p|$ es el *error absoluto* de la aproximación,
- $\frac{|\tilde{p} - p|}{|p|}$, $p \neq 0$; es el *error relativo* de la aproximación.

El error relativo nos da un porcentaje del error (si lo multiplicamos por 100) *y es “una medida” del número de dígitos correctos en la aproximación*. En general, el error relativo es más significativo que el error absoluto si tratamos con números muy pequeños o números muy grandes y nos previene de juzgar mal la exactitud de una aproximación en los casos en los que las escalas (números muy grandes o muy pequeños) nos podrían inducir a error.

Una combinación de ambas aproximaciones puede ser

- $\frac{|\tilde{p} - p|}{|p| + 1}$

Ejemplo 2.4

- a.) Sea $p = 0.112535 \times 10^{-6}$, una aproximación podría ser $\tilde{p} = 0.22507 \times 10^{-8}$. El error absoluto y el relativo son

$$|\tilde{p} - p| = 9.00281 \times 10^{-8} \quad \text{y} \quad \frac{|\tilde{p} - p|}{|p|} = 0.8$$

El error relativo evidencia de una manera más clara que los números “no son muy parecidos”, es decir, que el error en la aproximación es porcentualmente grande.

- b.) Sea $p = 0.8886110521 \times 10^7$, una aproximación podría ser $\tilde{p} = 0.8886110517 \times 10^7$. El error absoluto y el relativo son

$$|\tilde{p} - p| = 0.004 \quad \text{y} \quad \frac{|\tilde{p} - p|}{|p|} = 0.450141 \times 10^{-9}$$

El error absoluto dice que son números escasamente cercanos mientras que el relativo evidencia de una manera más clara que la aproximación tiene cerca de nueve dígitos correctos!

Ejemplo 2.5

En la tabla que sigue tenemos una lista de números y una aproximación. Observe como el relativo nos da “una medida” del número de dígitos correctos en la aproximación.

p	\tilde{p}	Error absoluto	Error relativo (%)	$\frac{ \tilde{p} - p }{ p + 1}$
0.000012	0.000009	0.000003	25 %	
4.5×10^{-7}	6.0×10^{-11}	4.4994×10^{-7}	99.98667 %	
10000000	1000000	9000000	90 %	
0.123456	0.1234	0.000056	0.0453603 %	

Para determinar el error absoluto y el error relativo, deberíamos conocer el valor exacto y el valor aproximado. En la práctica de los métodos numéricos, no conocemos *a priori* el valor exacto que estamos buscando pero si algunas aproximaciones. A veces tenemos una aproximación del error absoluto o a veces solo unas cuantas aproximaciones; en este último caso, *la estimación del error relativo se hace usando la mejor estimación disponible de p* .

Ejemplo 2.6

Sea $x_0 = 1$ y $x_{n+1} = \frac{1}{2} \left(x_n + \frac{2}{x_n} \right)$. Se sabe que $\lim_{n \rightarrow \infty} x_n = \sqrt{2}$. Podemos aproximar $\sqrt{2}$ usando esta sucesión.

La estimación del error relativo se hace *usando a x_{n+1} como la mejor estimación del valor exacto* (pues la sucesión es decreciente). En la tabla que sigue, comparamos la estimación del error relativo con el error relativo exacto.

	Error relativo estimado	Error relativo exacto
$x_0 = 1$	$ x_n - x_{n+1} / x_{n+1} $	$ \sqrt{2} - x_{n+1} / \sqrt{2} $
$x_1 = 1.5$	0.333333	0.0606602
$x_2 = 1.41666666666666666667$	0.0588235	0.00173461
$x_3 = 1.4142156862745098039$	0.0017331	$1.5018250929351827 \times 10^{-6}$
$x_4 = 1.4142135623746899106$	$1.5018239652057424 \times 10^{-6}$	$1.1276404038266872 \times 10^{-12}$
$x_5 = 1.4142135623730950488$	$1.1277376112344212 \times 10^{-12}$	0.

Podemos decir que $\sqrt{2} \approx 1.4142135623730950488$ con error relativo $\leq 1.1277376112344212 \times 10^{-12}$.

2**Ejercicios**

- 2.1** si $p = 0.001234$ y $\tilde{p} = 0.001234 \pm 0.000006$, verifique que \tilde{p} tiene 4 decimales correctos y solo dos dígitos significativos.
- 2.2** Si se sabe que $\tilde{p} = 4.6565434$ aproxima a un número p con $|p - \tilde{p}| \leq 0.000001$, entonces como mínimo ¿cuántos decimales exactos habría en la aproximación?
- 2.3** Encuentre un valor de tolerancia δ tal que si $|p - \tilde{p}| \leq \delta$, podamos garantizar que \tilde{p} tendrá al menos tres decimales exactos.

2.2 Aritmética del computador.

La representación de un número real en coma flotante requiere una base β y una precisión p . Por ejemplo, si $\beta = 10$ y $p = 3$ entonces 0.1 se representa como 1.00×10^{-1} .

El número

$$\pm \left(d_0 + \frac{d_1}{\beta^1} + \frac{d_2}{\beta^2} + \dots + \frac{d_{p-1}}{\beta^{p-1}} \right) \times \beta^e, \quad 0 \leq d_i \leq \beta - 1 \quad (2.1)$$

se representa en coma flotante como

$$\pm d_0.d_1d_2\dots d_{p-1} \times \beta^e, \quad 0 \leq d_i < \beta \quad (2.2)$$

$d_0.d_1d_2\dots d_{p-1}$ se llama “mantisa” y tiene p dígitos.

Ejemplo 2.7

En base 10

$$\begin{aligned} 3213 &= \left(\frac{3}{10} + \frac{2}{10^2} + \frac{1}{10^3} + \frac{3}{10^4} \right) \times 10^4 &= 0.3213 \times 10^4 \\ &= 0.03213 \times 10^5 \end{aligned}$$

Ejemplo 2.8

$$\begin{aligned}
 3.125 &= 2^0 + 2^1 + \frac{1}{2^3} = (11.001)_2 \times 2^0 \\
 &= \left(2 + \frac{1}{2^1} + \frac{1}{2^4}\right) \times 2^1 = (1.1001)_2 \times 2^1 \\
 &= \left(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^5}\right) \times 2^2 = (0.11001)_2 \times 2^2
 \end{aligned}$$

Representación en el computador. Si en un computador los números reales son representados en base 2, estos números se pueden almacenar como una sucesión de bits. Por ejemplo $3.125 = 1.1001 \times 2^1$ se almacena en 64 bits como

Cuenta con 15 decimales. Para un número representado en 64 bits, hay 11 bits para el exponente (± 308 en base 10) y 52 bits para la mantisa, esto nos da entre 15 y 17 dígitos decimales de precisión. e_{min} y e_{max} indican el exponente mínimo y el exponente máximo.

Cuando usamos la aritmética del computador, un número real se representa en doble precisión (double) con 15 decimales.

La comparación debe ser sencilla. Aunque los exponentes pueden ser negativos, se introduce un sesgo de tal manera que siempre queden como números mayores o iguales a cero (así, el exponente negativo más pequeño pasa a ser cero y los otros exponentes pasan a ser positivos). Esto se hace con el propósito de que la comparación de los números sea sencilla. En 64 bits el sesgo es 1023. Esta es la razón por la que en (2.3) el exponente es $(1000000000)_2 = 1023 + 1$.

Cómo lograr la unicidad. La representación en coma flotante no es única. Por ejemplo, 0.1 se puede representar como 0.01×10^1 o como 1.00×10^{-1} . Si en (2.2) pedimos que $d_0 \neq 0$, la representación se dice *normalizada*. La representación de 0.1 como 1.00×10^{-1} es la representación normalizada.

La representación normalizada resuelve la unicidad, pero hay un problema con el cero. Esto se resuelve con una convención para representarlo: $1.0 \times e^{e_{min}-1}$. Esto nos resta un exponente en la representación de los números pues $e_{min}-1$ se reserva para el cero.

Los números que podemos representar. En la representación normalizada en 64 bits, se sobrentiende que $d_0 = 1$, lo que nos deja 53 bits para la mantisa.

$$x = (-1)^s 1.b_1 b_2 \dots b_{52} \times 2^e \quad \text{con} \quad -1022 \leq e \leq 1023$$

El valor más grande sería $1.11\dots 11 \times 2^{1023} \approx 1.79 \times 10^{308}$ y el valor (normal¹) más pequeño sería $2.22\dots \times 10^{-308}$.

Excepciones. Si los cálculos exceden el máximo valor representable caemos en un estado de “sobrefljo” (overflow). En el caso del mínimo valor caemos en un estado de “subfljo” (underflow). En el estándar IEEE754, se reservan algunos patrones de bits para “ excepciones”. Hay un patrón de bits para números que exceden el máximo número representable: Inf, -Inf, NaN (not a number). Por ejemplo, 1./0. produce Inf. Si un cero es producido por un subfljo debido a negativos muy pequeños, se produce -0 y 1./(-0.) produce - Inf. NaN es producido por cosas tales como 0./0., $\sqrt{-2}$ o Inf - Inf.

■ **Redondeo.** Muchos números racionales tienen un cantidad infinita de decimales en su representación en base 10 y en base 2. Por ejemplo $1/3 = 0.333333\dots$. Esto también sucede en la representación en base 2. Por ejemplo,

$$0.2 = (0.00110011001100110\dots)_2.$$

Se debe hacer un corte para representar estos números en el computador. Si $rd(x)$ es la representación de x (por redondeo simétrico) en doble precisión, entonces

$$\left| \frac{x - rd(x)}{x} \right| \leq 2^{-52} \leq 0.5 \times 10^{-15}$$

2.3 Cancelación

La cancelación ocurre cuando se hace sustracción de dos números muy cercanos. Cuando se forma la resta $a - b$, se representan con el mismo exponente q y algunos dígitos significativos en la mantisa son cancelados y al normalizar se mueven dígitos a la izquierda y se disminuye el exponente. Al final de la mantisa aparecen ceros inútiles.

Ejemplo 2.9

Consideremos la función $f(x) = x^7 - 7 * x^6 + 21 * x^5 - 35 * x^4 + 35 * x^3 - 21 * x^2 + 7 * x - 1 = (x - 1)^7$. Aquí podemos tener un fenómeno de cancelación.

```
p = function(x) x^7 - 7*x^6 + 21*x^5 - 35*x^4 + 35*x^3 - 21*x^2 + 7*x - 1
f = function(x) (x - 1)^7

x0 = 0.99
cat("f(0.99) = ", f(x0), "      ", "p(0.99) = ", p(x0))
#f(0.99) = -1.00000000000006e-14      p(0.99) = -1.4210854715202e-14
```

El error relativo entre $f(0.99)$ y $p(0.99)$ es grande, $\approx 42\%$. Para detectar en qué parte del cálculo se magnifica este error, sepáramos el cálculo,

¹En aritmética de punto flotante, los números que son más pequeños que el más pequeño número ‘normal’ que se puede representar, se llaman “subnormales”. Los números subnormales se introducen para preservar la importante propiedad $x = y \iff x - y = 0$. Cuando se alcanza el mínimo normal, la mantisa se va llenando con ceros permitiendo la representación de números más pequeños. El subnormal mínimo sería 4.9×10^{-324} .

```
x = 0.99
y = c(x^7, - 7*x^6, 21*x^5, - 35*x^4, 35*x^3, - 21*x^2, 7*x, - 1)
cumsum(y)
#[1] 9.320653479069899e-01 -5.658295697900010e+00
#[3] 1.431249534999999e+01 -1.930836500000001e+01
#[5] 1.465209999999998e+01 -5.9300000000000015e+00
#[7] 9.99999999999848e-01 -1.521005543736464e-14
```

Cuando vamos sumando hasta $x^7 - 7 * x^6 + 21 * x^5 - 35 * x^4 + 35 * x^3 - 21 * x^2 + 7 * x$, la suma es 0.9999999999999848, pero al restar 1 se da el fenómeno de cancelación en $p(x)$, en vez de obtener **-1.00000000000006e-14** obtenemos **-1.4210854715202e-14**.

Ejemplo 2.10

Se sabe que $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. Ya habíamos visto la implementación

```
options(digits = 10)
funexp = function(x, n) 1+sum(cumprod(x/1:n)) #=sum(x/1, x/1*x/2, x/1*x/2*x/3,...)
funexp(1,20)
#[1] 2.718281828
```

Pero si $x < 0$ los términos de la serie cambian de signo y si $|x|$ es grande, la función **funexp** deberá retornar valores pequeños en presencia de algunos sumando muy grandes. En algún momento estaremos cerca de un fenómeno de cancelación cuando sumemos “el siguiente” término. La manera de arreglar el problema es usar el hecho de que $e^{-x} = \frac{1}{e^x}$

```
cat(funexp(-20, 100), " ", exp(-20)) # no hay mejora si n>100
#[1] -3.8696e-09 -2.061154e-09
# --- e^-x = 1/e^x
cat(1/funexp(20, 100), " ", exp(-20))
#[1] 2.061154e-09 2.061154e-09
```

Ejemplo 2.11

Por ejemplo, usando aritmética con diez decimales, si $a = \sqrt{9876} = 9.937806599 \times 10^1$ y $b = \sqrt{9875} = 9.937303457 \times 10^1$, entonces $a - b = 0.000503142 \times 10^1$. Normalizando se obtiene,

$$\sqrt{9876} - \sqrt{9875} = 5.031420000 \times 10^{-3}$$

y perdemos dígitos significativos. Se puede arreglar el problema *racionalizando*:

$$\sqrt{a} - \sqrt{b} = \frac{a - b}{\sqrt{a} + \sqrt{b}},$$

esto cambia la resta (de números cercanos) $\sqrt{9876} - \sqrt{9875}$ por la suma $\sqrt{9876} + \sqrt{9875}$ y la resta $9876 - 9875$ no presenta problemas.

$$\sqrt{9876} - \sqrt{9875} = \frac{9876 - 9875}{\sqrt{9876} + \sqrt{9875}} = 5.031418679 \times 10^{-3}.$$

Es usual usar algunos trucos para minimizar este fenómeno de cancelación.

(a.) Cambiar $\sqrt{a} - \sqrt{b}$ por $\frac{a - b}{\sqrt{a} + \sqrt{b}}$

(b.) Cambiar $\sin a - \sin b$ por $2 \cos \frac{a+b}{2} \sin \frac{a-b}{2}$

(c.) Cambiar $\log a - \log b$ por $\log(a/b)$

(d.) Si f es suficientemente suave y h pequeño, podemos cambiar $y = f(x+h) - f(x)$ por la expansión de Taylor

$$y = f'(x)h + 0.5f''(x)h^2 + \dots$$

Los términos en esta serie decrecen rápidamente si h es suficientemente pequeño, así que el fenómeno de cancelación deja de ser un problema.

A veces, dieciséis dígitos no bastan. Cuando tratamos con números de máquina, hay un error que se va propagando. Esta propagación del error es, en algunos casos, muy dañina. A veces el daño es producto del fenómeno de *cancelación*, al restar números parecidos y muy pequeños y a veces es por problemas de inestabilidad del algoritmo en curso. Algunos cálculos podemos mejorarlo aumentando la precisión, como se muestra en los ejemplos que siguen,

Ejemplo 2.12

Consideremos la cuadrática $P(x) = ax^2 + bx + c$ donde $a = 94906265.625$, $b = -189812534$ y $c = 94906268.375$. Con dieciséis dígitos se obtiene

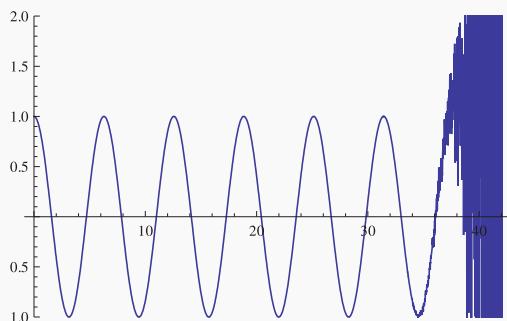
$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = 1.000000014487979$$

Pero $P(x_2) = -0.00000002980232!$. Mmmmm, aquí los números que son cercanos son b^2 y $4ac$ y no es de mucha utilidad racionalizar. Todavía nos queda la opción de trabajar con más decimales. En **R** hay un paquete para trabajar con multiprecisión (el paquete **Rmpfr**^a).

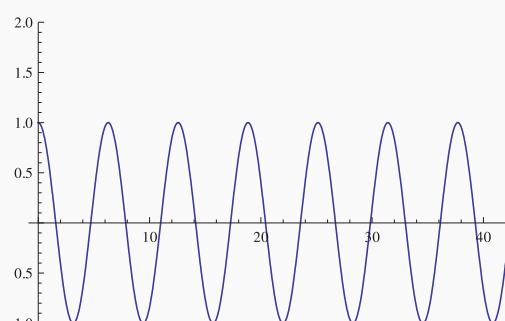
^aEn Ubuntu, este paquete requiere instalar “C++ wrapper for the GNU MPFR C library” (desde el Centro de Software) y ‘GMP’(Multiple Precision library). Posiblemente solo necesite instalar, usando la terminal: sudo apt-get install libmpfr-dev

Ejemplo 2.13

Representación gráfica de $f(x) = \sum_{n=0}^{200} \frac{(-1)^n x^{2n}}{(2n)!}$. Aquí también los cálculos se ven afectados si trabajamos con poca precisión.



Precisión = 15



Precisión = 200

2.4 Propagación del Error

Cuando un error de redondeo ha sido introducido, este se suma a otros errores y se propaga. Supongamos que queremos calcular el valor $f(x) \in \mathbb{R}$. En el computador x es aproximado con un número racional \tilde{x} , así que $\tilde{x} - x$ es el *error inicial* y $\epsilon_1 = f(\tilde{x}) - f(x)$ es el correspondiente *error propagado*. En muchos casos, en vez de f se usa una función más simple f_1 (a menudo una expansión *truncada* de f). La diferencia $\epsilon_2 = f_1(\tilde{x}) - f(\tilde{x})$ es el *error de truncación*. Luego, las operaciones que hace el computador son “seudo-operaciones” (por el redondeo) por lo que en vez de $f_1(\tilde{x})$ se obtiene otro valor incorrecto $f_2(\tilde{x})$. La diferencia $\epsilon_3 = f_2(\tilde{x}) - f_1(\tilde{x})$ se podría llamar *el error propagado por los redondeos*. El error total sería

$$\epsilon = f_2(\tilde{x}) - f(x) = \epsilon_1 + \epsilon_2 + \epsilon_3$$

La aritmética usada por el computador no respeta la aritmética ordinaria. Cada simple operación en punto flotante casi siempre genera un error pequeño que se puede propagar en las siguientes operaciones. Se puede minimizar los errores de redondeo incrementando el número de cifras significativas en el computador y así el error no será en general muy dañino excepto en casos particulares, por ejemplo en los que se restan cantidades de signo opuesto y muy parecidas en valor absoluto.

■ **Análisis del error de propagación.** El análisis de error es importante cuando se quiere investigar y/o garantizar el desempeño de los métodos numéricos usados en problemas teóricos y prácticos. Ejemplos catastróficos producidos por errores de redondeo se puede ver en <http://mathworld.wolfram.com/RoundoffError.html>.

Sea $\rho = \left| \frac{x - \text{rd}(x)}{x} \right|$. Entonces $\text{rd}(x) = x(1 + \varepsilon)$, con $|\varepsilon| = \rho$, $|\varepsilon| \leq \text{eps}$

Para seguir adelante necesitamos definir un *modelo* para la aritmética de la computadora. Excepto por la ocurrencia de sobreflujo o subflujo, vamos a suponer en nuestro modelo que las operaciones $+, -, \cdot, /$ producen un resultado

redondeado que es *representable* en el computador. Denotamos con $fl(x+y)$, $fl(x \cdot y)$, ... el resultado de estas operaciones (el resultado que produce la máquina). Entonces, por ejemplo

$$fl(x \cdot y) = x \cdot y (1 + \varepsilon), \quad |\varepsilon| \leq \text{eps}$$

Ahora bien, nuestro interés es analizar el error en los resultados causados por errores en los datos. Vamos a suponer que $x(1 + \varepsilon_x)$ y $y(1 + \varepsilon_y)$ son valores de x e y contaminados con errores relativos ε_x y ε_y . Analicemos el error relativo en cada operación $\cdot, +, -, /$.

- *Multiplicación.* Supongamos que ε_x y ε_y son tan pequeños que los términos de orden dos ε_x^2 , ε_y^2 , $\varepsilon_x \cdot \varepsilon_y$, ... u orden superior, puedan ser despreciados (respecto a los ε' s), por ejemplo ε_x y ε_y podrían ser los errores relativos en la representación. Entonces

$$x(1 + \varepsilon_x) \cdot y(1 + \varepsilon_y) \approx x \cdot y(1 + \varepsilon_x + \varepsilon_y)$$

y por tanto, el error relativo $\varepsilon_{xy} \approx \varepsilon_x + \varepsilon_y$. Por ejemplo, si uno suma mil números con errores cada uno del orden 10^{-10} , entonces el error acumulado sería de $10^{-10} * 1000 = 10^{-7}$

Así, en general, en el producto los errores relativos de los datos se suman en el resultado. Esta situación la vamos a considerar *aceptable*.

- *División.* Si $y \neq 0$ entonces, usando la expansión en serie de $1/(x+1)$,

$$\frac{x(1 + \varepsilon_x)}{y(1 + \varepsilon_y)} = \frac{x}{y} (1 + \varepsilon_x)(1 - \varepsilon_y + \varepsilon_y^2 - \dots) \approx \frac{x}{y} (1 + \varepsilon_x - \varepsilon_y)$$

y entonces el error relativo $\varepsilon_{x/y} \approx \varepsilon_x - \varepsilon_y$ el cual es aceptable.

- *Suma y resta.* Como x, y pueden tener cualquier signo, basta con considerar la suma.

$$\begin{aligned} x(1 + \varepsilon_x) + y(1 + \varepsilon_y) &= x + y + x\varepsilon_x + y\varepsilon_y \\ &= (x + y) \left(1 + \frac{x\varepsilon_x + y\varepsilon_y}{x + y}\right) \end{aligned}$$

si $x + y \neq 0$. Entonces

$$\varepsilon_{x+y} = \frac{x}{x+y} \varepsilon_x + \frac{y}{x+y} \varepsilon_y$$

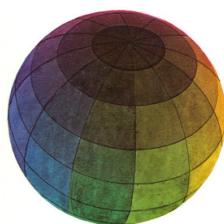
que es nuevamente una combinación lineal de los errores en los datos.

- Si x e y son de *igual signo* entonces $0 \leq \frac{x}{x+y} \leq 1$, luego

$$\varepsilon_{x+y} \leq |\varepsilon_x| + |\varepsilon_y|$$

que es un resultado aceptable.

- Si x e y son de *signo contrario* entonces $\frac{x}{x+y}$ y $\frac{y}{x+y}$ pueden ser números arbitrariamente grandes cuando $|x+y|$ es arbitrariamente pequeño comparado con $|x|$ e $|y|$. Y esto ocurre cuando x e y son casi iguales en valor absoluto, pero de signo contrario. Este fenómeno es el llamado *error de cancelación*. Es un talón de Aquiles del análisis numérico y debe ser evitado siempre que sea posible. Observe que los efectos de la “cancelación” se pueden alcanzar por la suma de pequeñas dosis de cancelación en grandes cálculos.



Revisado: Marzo, 2016

Versión actualizada de este libro:

<https://tecdigital.tec.ac.cr/revistamatematica/Libros/>



3 — Polinomio de Taylor

En cálculo numérico, muchos resultados se derivan de la aplicación del teorema de Taylor.

Teorema 3.1 (Teorema de Taylor con resto)

Sea f una función con sus primeras $n+1$ derivadas continuas en $[a, b]$ y sean $x, x_0 \in [a, b]$. Entonces,

$$f(x) = P_n(x) + R_n(x)$$

donde

$$P_n(x) = \sum_{k=0}^n \frac{f^{(n)}(x_0)}{k!} (x - x_0)^k$$

y

$$R_n(x) = \frac{1}{n!} \int_{x_0}^x (x-t) f^{(n+1)}(t) dt$$

También, existe ξ_x entre x y x_0 tal que

$$R_n(x) = \frac{(x-x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi_x)$$

Las dos expresiones para el resto $R_n(x)$ dan el mismo resultado y la segunda expresión se puede deducir de la primera usando el teorema del valor medio para integrales.

El teorema de Taylor nos da una representación de una función usando un polinomio y una expresión para el

error. En métodos numéricos esto nos permite reemplazar una función por algo más sencillo, un polinomio y una estimación del error.

Ejemplo 3.1

Veamos algunas expansiones en serie de Taylor usando alrededor de $x_0 = 0$.

a.) $e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \dots + \frac{1}{n!}x^n + \frac{1}{(n+1)!}x^{n+1}e^{\xi_x}$ con ξ_x entre 0 y x

b.) $\sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 + \dots + \frac{(-1)^n}{(2n+1)!}x^{2n+1} + \frac{(-1)^{n+1}}{(2n+3)!}x^{2n+3}\cos\xi_x$ con ξ_x entre 0 y x

c.) $\cos(x) = \sum_{k=0}^n \frac{(-1)^k}{(2k)!}x^{2k} + \frac{(-1)^{n+1}}{(2n+2)!}x^{2n+2}\cos\xi_x$ con ξ_x entre 0 y x

3.1 Estimación del error

Como $f(x) = P_n(x) + R_n(x)$ entonces $f(x) \approx P_n(x)$ y el error de esta aproximación sería

$$\text{Error} = |f(x) - P_n(x)| = \left| \frac{(x-x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi_x) \right| \quad \text{con } \xi_x \text{ entre } 0 \text{ y } x$$

En general, no podemos determinar ξ_x por lo que solo podríamos tener una **estimación del error**. Una manera de obtener esta estimación es usando el máximo absoluto M_{n+1} de $|f^{(n+1)}|$ en algún intervalo cerrado que contenga a 0 y x (y por tanto a ξ_x).

$$\text{Como } \left| \frac{(x-x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi_x) \right| \leq \left| \frac{(x-x_0)^{n+1}}{(n+1)!} M_{n+1} \right| \quad \text{entonces} \quad \text{Error} \leq \left| \frac{(x-x_0)^{n+1}}{(n+1)!} M_{n+1} \right|$$

Primero recordemos como determinar los máximos y mínimos absolutos de una función continua y derivable en un intervalo $[a, b]$.

- $f \in C^n[a, b]$ indica que f tiene derivadas continuas hasta el orden n en $[a, b]$. En particular, $f \in C[a, b]$ indica que f es continua en $[a, b]$ y $f \in C^1[a, b]$ indica que f es derivable en $[a, b]$
- Recordemos los puntos críticos en $[a, b]$ de $f \in C^1[a, b]$ son las soluciones de la ecuación $f'(x) = 0$ en $[a, b]$. Si los puntos críticos en $[a, b]$ son $\{x_0, x_1, \dots, x_k\}$ entonces el **máximo** y el **mínimo** valor del conjunto

$$\{f(a), f(b), f(x_0), f(x_1), \dots, f(x_k)\}$$

corresponden al *máximo y el mínimo absoluto* de esta función en $[a, b]$.

- Si f es una función de una variable y si $f(x^*) = 0$ entonces decimos que x^* es un *cero* de f . Si f es un polinomio también se dice que x^* es una *raíz*.

Ejemplo 3.2

Una primera pregunta que nos interesa es esta: ¿Para aproximar e^x con un polinomio de Taylor alrededor de $x_0 = 0$, con $x \in [-1, 1]$, que grado n debemos escoger?

El problema es: Determine $n \in \mathbb{N}$ tal que si aproximamos e^x con el polinomio de Taylor $P_n(x)$, en $[-1, 1]$, entonces $\text{Error} \leq 0.5 \times 10^{-7}$.

Solución: Como $\text{Error} \leq \left| \frac{x^{n+1}}{(n+1)!} M_{n+1} \right|$ con M_{n+1} el máximo de $|f^{(n+1)}|$ en $[-1, 1]$.

Bien, hay que calcular M_{n+1}

Primero: Los puntos críticos de $f^{(n+1)}(x) = e^x$. Como $f^{(n+2)}(x) = e^x$ no se anula, entonces $f^{(n+1)}$ no tiene puntos críticos.

Segundo: Comparar para obtener el máximo absoluto. $M_{n+1} = \max\{|f^{(n+1)}(-1)|, |f^{(n+1)}(1)|\} = e^1$

Finalmente: Necesitamos $n \in \mathbb{N}$ tal que $\text{Error} \leq \left| \frac{x^{n+1}}{(n+1)!} e \right| \leq 0.5 \times 10^{-7}$. Como $x \in [-1, 1]$ entonces $|x^n| \leq 1$, entonces

$$\text{Error} \leq \left| \frac{1}{(n+1)!} e \right| \leq 0.5 \times 10^{-7}$$

Por tanto, observamos que $\text{Error} \leq \left| \frac{1}{(n+1)!} e \right| \leq 0.5 \times 10^{-7}$ desde $n = 11$ en adelante.

Por tanto, $e^x \approx 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \dots + \frac{1}{11!}x^{11}$ con error $\leq 0.5 \times 10^{-7}$ si $x \in [-1, 1]$.

Ejemplo 3.3

Estime el error cometido al aproximar $f(x) = \sin x$, con $x \in [-\pi/3, \pi/3]$ con un polinomio de Taylor, alrededor de $x_0 = 0$, de orden $n = 5$.

Solución: En este caso $f(x) = \sin x$. Para determinar el error estimado, debemos calcular el máximo absoluto de $|f^{(7)}(x)| = |\cos x|$ en $[-\pi/3, \pi/3]$.

Primero: Puntos críticos. $|f^{(8)}(x)| = |\sin x| = 0 \implies x = 0 + 2k\pi$ con $k \in \mathbb{Z}$. El único punto crítico en $[-\pi/3, \pi/3]$ sería $x = 0$.

Segundo: Comparación. $M_7 = \max\{|f^{(7)}(-\pi/3)|, |f^{(7)}(\pi/3)|, |f^{(7)}(0)|\} = 1$

$$\begin{aligned} R_5(x) &\leq \left| \frac{x^{5+1}}{(5+1)!} M_{5+1} \right| \\ &\leq \left| \frac{x^6}{6!} M_6 \right| \\ &\leq \left| \frac{(\pi/3)^6}{6!} 1 \right| \quad \text{pues } M_6 = 1 \quad y \quad x^6 \leq (\pi/3)^6 \quad (y = x^6 \text{ es creciente}) \end{aligned}$$

$$R_5(x) \leq 0.001831636$$

Por tanto $\sin(x) \approx x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5$ con error ≤ 0.001831636 en $[-\pi/3, \pi/3]$

Ejercicio 3.1 Para cada una de las funciones que siguen, determine el polinomio de Taylor de orden $n = 3$ alrededor de $x_0 = 0$ y estime el error de la aproximación en el intervalo que se indica.

a.) $f(x) = e^{-x}$ con $x \in [0, 1]$

b.) $f(x) = \ln(1+x)$ con $x \in [-0.5, 0.5]$

c.) $f(x) = \sqrt{1+x^2}$ con $x \in [0, 1]$

Ejercicio 3.2 Para cada una de las funciones que siguen, determine el orden n del polinomio de Taylor, alrededor de $x_0 = 0$, de tal manera que el error de la aproximación en el intervalo indicado sea $\leq 0.5 \times 10^{-7}$.

a.) $f(x) = e^{-x}$ con $x \in [0, 1]$

b.) $f(x) = \ln(1-x)$ con $x \in [-0.5, 0.5]$

c.) $f(x) = \sqrt{1+x}$ con $x \in [0, 1]$

3.1.1 Implementación

En la base de R tenemos la función `D()` para calcular la derivada simbólica de una función. La derivada n -ésima se puede calcular con una función `DD()` sugerida en la “ayuda” de la función `D`. Sin embargo, podemos usar el paquete `pracma`. La función `taylor(f, x0, n = 4, ...)` devuelve los coeficientes del polinomio de Taylor de f , orden $n=4$, alrededor de $x0$.

■ Código R 3.1: Polinomio de Taylor con “pracma”

```
require(pracma)
f = function(x) log(1+x)
p = taylor(f, 0, 4) # Polinomio de Taylor de orden 4, alrededor de a=0.
p
# Coeficientes
# [1] -0.250004  0.333334 -0.500000  1.000000  0.000000
# Evaluar en x=0.1
polyval(p, 0.1)
# [1] 0.09530833
log(1+0.1)
# [1] 0.09531018
```

Ejercicio 3.3 Aproximación de una función con un polinomio de Taylor alrededor de $a = 0$ y estimación del error.

Considere la función $f(x) = e^x \cos x$.

a.) Obtenga el polinomio de Taylor $P_3(x)$ (de orden $n = 3$) para f alrededor de $a = 0$.

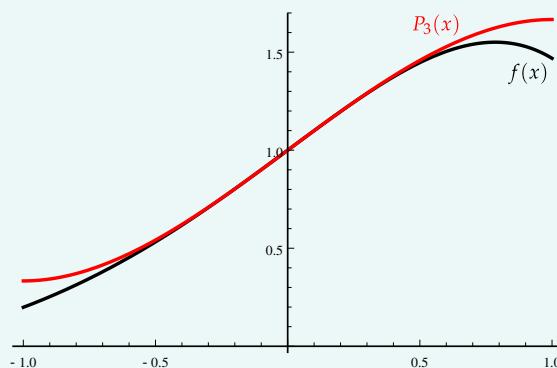


Figura 3.1: Función f en negro y el polinomio de Taylor $P_3(x)$ alrededor de $a = 0$, en rojo

- b.) Aproximar $f(0.3)$ con el polinomio $P_3(x)$ y estimar el error cometido en esta aproximación, usando una estimación del error $R_3(x)$, tal y como se hizo en clase.

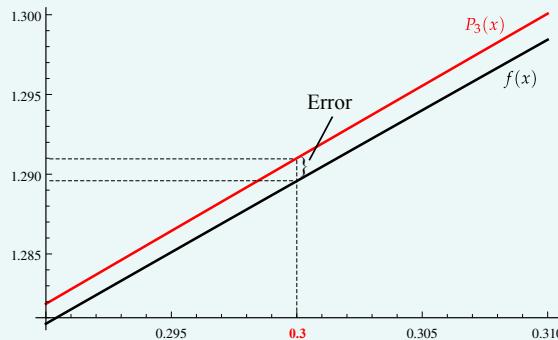
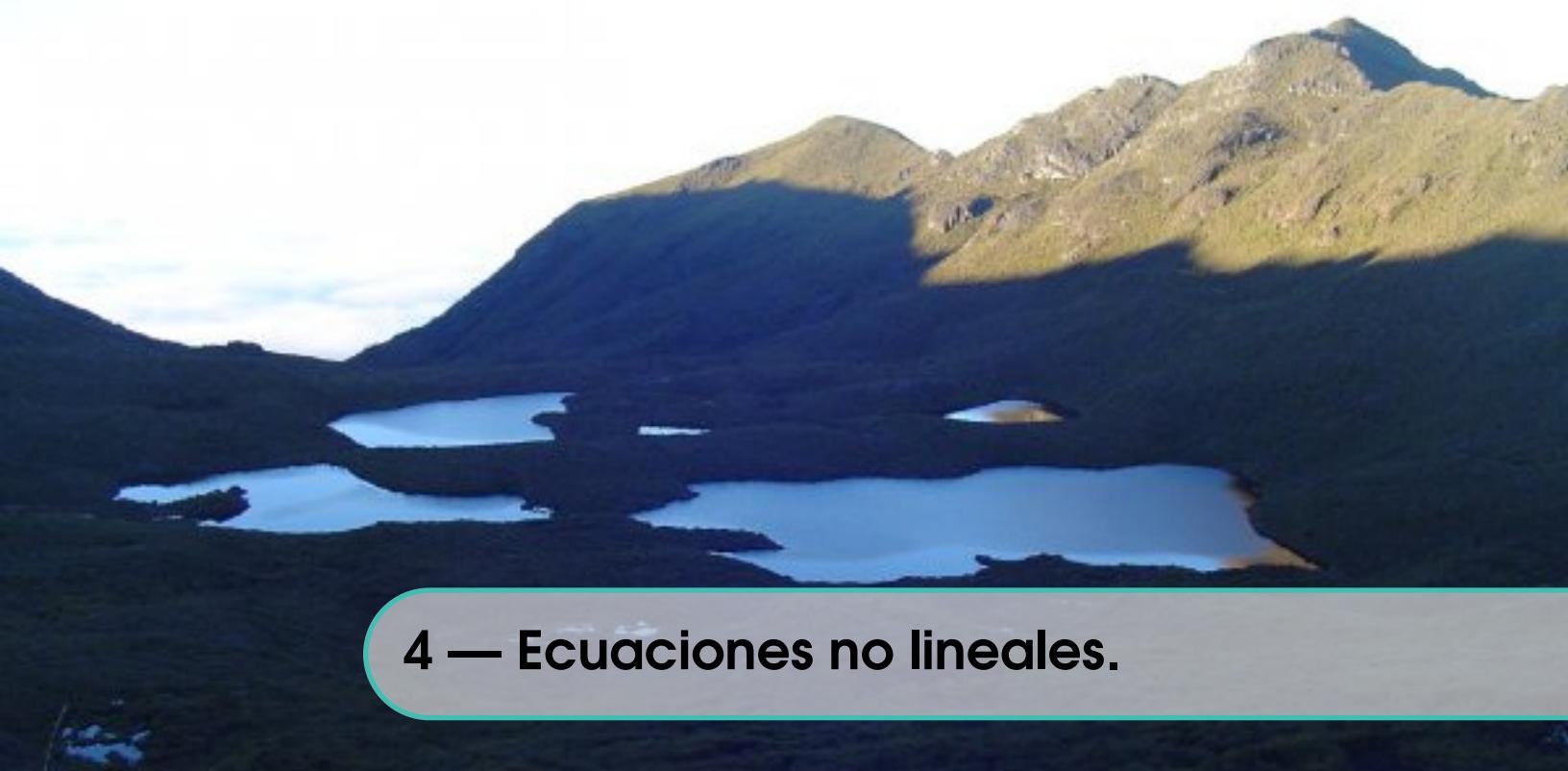


Figura 3.2: Diferencia entre el valor calculado por el polinomio de Taylor (en rojo) y el valor de la función



4 — Ecuaciones no lineales.

En general, no es posible determinar los *ceros* de una función, es decir, valores x^* tal que $f(x^*) = 0$, en un número finito de pasos. Tenemos que usar métodos de aproximación. Los métodos son usualmente iterativos y tienen la forma: Iniciando con una aproximación inicial x_0 (o un intervalo $[a, b]$), se calculan aproximaciones sucesivas x_1, x_2, \dots y elegimos x_n como aproximación de x^* cuando se cumpla un *criterio de parada dado*. A los ceros de un polinomio se les conoce también como *raíces*.

En este capítulo veremos los métodos iterativos usuales: bisección, regula falsi, punto fijo, Newton y el método de la secante. En general, no usamos estos métodos de manera aislada sino más bien combinada. Por eso se incluyen secciones con métodos híbridos. El método de bisección es muy confiable, pero relativamente lento. Los métodos de Newton y la secante son más veloces, pero no son tan confiables como bisección. Bisección es óptimo para funciones continuas (en general) pero no para funciones derivables o convexas. En este último caso, el método de Newton es veloz, pero necesita el cálculo de la derivada (no todas las funciones derivables tienen derivadas que se pueden expresar en términos de funciones elementales o funciones especiales) y podría colapsar si la derivada toma valores muy pequeños en el proceso. En las funciones obtenidas por interpolación corre el riesgo de caer en un ciclo. El método de la secante es veloz y no requiere la derivada, sino una aproximación. Aún así, corre riesgos inherentes al comportamiento de la derivada en las cercanías de la raíz.

El método de Dekker-Brent combina la confiabilidad de bisección con la velocidad del método de la secante y el método de interpolación cuadrática inversa. Iniciando con un intervalo donde la función cambia de signo, el siguiente paso toma el camino más veloz disponible (secante o interpolación cuadrática inversa, si no hay peligro de colapso) sin dejar nunca el intervalo donde hay cambio de signo. Así en el peor de los casos, en el siguiente paso se usaría bisección. Este algoritmo es un método de tipo adaptativo, está balanceado de tal manera que siempre encuentra una respuesta y es siempre más rápido que bisección. Algunos paquetes de software, como *Mathematica*® y *MatLab*®, usan Newton, secante y el método de Brent para aproximar ceros de funciones. Para encontrar las raíces de un polinomio se usan algoritmos especializados, por ejemplo el algoritmo de Jenkins-Traub.

4.1 Orden de convergencia

El orden de convergencia nos da una 'medida' de la rapidez con la que una sucesión de aproximación converge, en particular nos podría informar con qué rapidez, a partir de cierto índice, ganamos cifras decimales correctas.

Definición 4.1 (Orden de convergencia).

Supongamos que $\lim_{n \rightarrow \infty} u_n = u^*$ y que $u_n \neq u^* \forall n$. Se dice que la sucesión tiene *orden de convergencia* $q \geq 1$ si

$$\lim_{n \rightarrow \infty} \frac{|u_{n+1} - u^*|}{|u_n - u^*|^q} = K \text{ para alguna constante } 0 < K < \infty. \quad (4.1)$$

Para entender lo que dice la definición, supongamos que δ_n denota el número de lugares decimales exactos en la aproximación en la n -ésima iteración, entonces

$$\delta_n \approx -\log_{10} |u_n - u^*|$$

Si $u_n = 0.123447$ y $u^* = 0.123457$ entonces $\delta_n \approx -\log_{10} |0.00001| = 5$

Si $u_n = 1.53222$ y $u^* = 1.5332423$ entonces $\delta_n \approx -\log_{10} |0.00001| = 2.99042$

Tomando logaritmos de base 10 a ambos lados del límite (4.1) podemos establecer que

$$\delta_{n+1} \approx q\delta_n - \log_{10} |K|.$$

Es decir, si el orden de convergencia es q , en la siguiente iteración (después de la iteración n -ésima), el número de lugares decimales exactos es aproximadamente $q\delta_n + L$ con $L = -\log_{10} |K|$.

- q no necesariamente es un entero.
- Si $q = K = 1$, u_n converge más despacio que una sucesión que converja linealmente y se dice que converge *sublinealmente*.
- Si (4.1) se da con $q = 1$ y $K = 0$, pero no con $q > 1$, la convergencia se dice *superlineal*.
- Si $q = 1$ y $0 < K < 1$, u_n se dice que *converge linealmente* y K es la *tasa* de convergencia.
- Si $q = 2$ entonces u_n se dice que *converge cuadráticamente* y se espera que a partir de algún subíndice N , cada iteración aproximadamente *duplica* el número de lugares decimales exactos. Observe que en el caso particular $q = 2$ y $K = 1$: $\delta_{n+1} \approx q\delta_n - \log_{10} |K| \approx 2\delta_n$.

Ejemplo 4.1

$q = 1$ indica que se gana una cifra decimal exacta cada cierto número promedio de iteraciones. La sucesión $x_n = \frac{1}{6} - \frac{1}{3} \left(\frac{-1}{2}\right)^n$ converge a $1/6 = 0.166666666...$

El orden de convergencia es $q = 1$. En efecto,

$$\lim_{n \rightarrow \infty} \frac{|u_{n+1} - 1/6|}{|u_n - 1/6|^q} = \lim_{n \rightarrow \infty} \frac{\frac{1}{3} \left(\frac{1}{2}\right)^{n+1}}{\left(\frac{1}{3}\right)^q \left(\frac{1}{2}\right)^{nq}} = \lim_{n \rightarrow \infty} 3^{1-q} \left(\frac{1}{2}\right)^{n(1-q)+1} = \begin{cases} \frac{1}{2} & \text{si } q = 1 \\ \infty & \text{si } q > 1. \end{cases}$$

En la tabla se observa que efectivamente $\delta_{n+1} \approx \delta_n$ y hay que esperar algunas iteraciones adicionales para ganar un nuevo decimal exacto.

n	δ_{n+1}	δ_n	x_n
40	12.8193775849834	12.5183078278654	0.1666666666666818
	13.1203280613792	12.8193775849834	0.1666666666666591
	13.4215171101422	13.1203280613792	0.1666666666666705
	13.7222290578374	13.4215171101422	0.166666666666648
	14.0238953825265	13.7222290578374	0.166666666666676
	14.3236536511268	14.0238953825265	0.16666666666662
	14.6272308358047	14.3236536511268	0.16666666666669
	14.9231813059394	14.6272308358047	0.16666666666665
	15.2344304667850	14.9231813059394	0.16666666666667
49	15.5152570763607	15.2344304667850	0.16666666666666666

Ejemplo 4.2 (Convergencia cuadrática).

Sea $x_0 > 0$ y $x_{n+1} = 0.5 \left(x_n + \frac{2}{x_n} \right)$. Se sabe que $\lim_{n \rightarrow \infty} x_n = \sqrt{2}$. En este caso, la convergencia es cuadrática: En efecto, primero veamos que $x_{n+1} - \sqrt{2} = (x_n - \sqrt{2})^2$, entonces

$$\frac{x_{n+1} - \sqrt{2}}{(x_n - \sqrt{2})^q} = \frac{(x_n - \sqrt{2})^2}{2x_n (x_n - \sqrt{2})^q} = \frac{(x_n - \sqrt{2})^{2-q}}{2x_n}$$

por tanto,

$$\lim_{n \rightarrow \infty} \left| \frac{(x_n - \sqrt{2})^{2-q}}{2x_n} \right| = \begin{cases} 0 & \text{si } q = 1 \\ \frac{1}{2\sqrt{2}} & \text{si } q = 2 \\ \infty & \text{si } q > 2 \end{cases}$$

Comparemos $\sqrt{2} = 1.4142135623730950...$ con los valores de la segunda columna de la tabla (4.1).

n	x_n	$ x_n - \sqrt{2} $
0	1.5	0.5
1	1. <u>4</u> 16666666666667	0.083333333
2	1. <u>41</u> 421568627451	0.00245098
3	1. <u>4142</u> 1356237469	2.1239×10^{-6}
4	1. <u>41421</u> 356237310	1.59472×10^{-12}
5	1. <u>414213</u> 56237309	2.22045×10^{-16}

La convergencia es muy rápida y se puede observar como *aproximadamente*, la precisión se duplica (en las cercanías de $\sqrt{2}$) en cada iteración.

Aunque en general no disponemos de un mecanismo analítico para resolver una ecuación arbitraria, si tenemos métodos para aproximar una solución mediante aproximaciones sucesivas. En lo que resta del capítulo vamos ver algunas de estos métodos, su confiabilidad y su velocidad de convergencia.

4.2 Método de Punto Fijo

En muchos casos una ecuación no lineal aparece en la forma de “*problema de punto fijo*”:

$$\text{Encuentre } x \text{ tal que } x = g(x) \tag{4.2}$$

Un número $x = x^*$ que satisface esta ecuación se llama *punto fijo de g*.

Ejemplo 4.3

- En el problema de punto fijo $x = \sin(x)$, tenemos $g(x) = \sin(x)$ y un punto fijo de g es $x^* = 0$.
- Si $g(x) = x^2$, los puntos fijos son $x = 0$ y $x = 1$ pues $0^2 = 0$ y $1^2 = 1$.
- Si $g(x) = \ln x + x$, un punto fijo es $x = 1$ pues $\ln 1 + 1 = 1$.

Ejemplo 4.4

¿Cuáles son los puntos fijos de $g(x) = \ln(x)$ o de $g(x) = \cos(x)$? Geométricamente, un punto fijo corresponde al valor de la abscisa donde la gráfica de $y = g(x)$ interseca a la recta $y = x$.

En el problema de punto fijo $x = \cos(x)$, tenemos $g(x) = \cos(x)$. El único punto fijo de g es $x^* \approx 0.7390851332151607\dots$

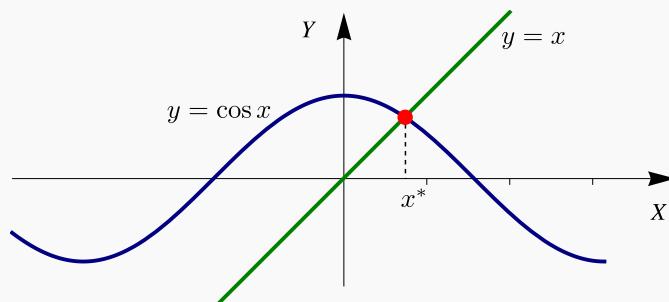


Figura 4.1: Punto fijo de $\cos(x)$: $x^* \approx 0.73908\dots$

Una ecuación $f(x) = 0$ se puede escribir en la forma (4.2) despejando x (si se pudiera). En este caso se obtiene $f(x) = 0 \implies x = g(x)$

Ejemplo 4.5

La ecuación $x^3 + x + 1 = 0$ se puede poner como un problema de punto fijo despejando x de varias maneras,

a.) $x = -x^3 - 1$. En este caso $g(x) = -x^3 - 1$

b.) $x = \sqrt[3]{-x - 1}$. En este caso $g(x) = \sqrt[3]{-x - 1}$

c.) $x = \frac{\sqrt{-1-x}}{x^2}$

d.) ...

■ **Iteración de punto fijo.** Un esquema iterativo de punto fijo define por recurrencia una sucesión $\{x_n\}$ de la siguiente manera:

$$\text{Aproximación inicial: } x_0 \in \mathbb{R}. \quad \text{Iteración de "punto fijo": } x_{i+1} = g(x_i), i = 0, 1, 2, \dots \quad (4.3)$$

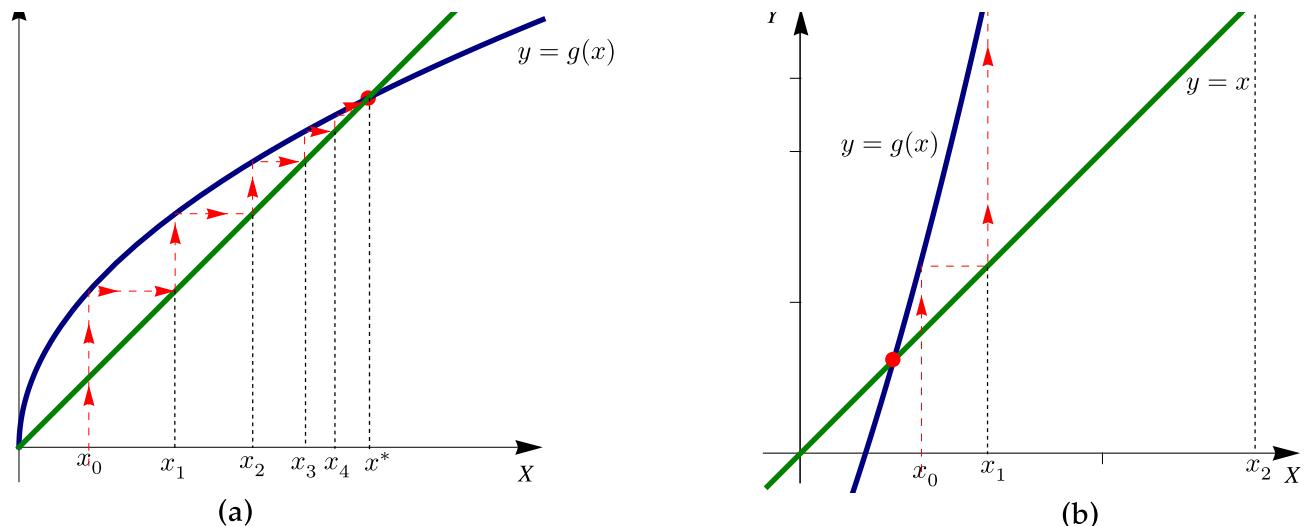


Figura 4.2: Iteraciones del método de punto fijo: convergencia y divergencia. Observe que $|g'(x)| < 1$ en los alrededores de $x = x^*$ en la figura (a)

Teorema 4.1

Sea la sucesión $\{x_n\}$ definida por $x_0 \in \mathbb{R}$, $x_{i+1} = g(x_i)$, $i = 0, 1, 2, \dots$ con g continua. Si $\lim_{n \rightarrow \infty} x_n = x^*$, entonces x^* es punto fijo de g .

Ejemplo 4.6

Consideremos la ecuación $x = \frac{x^4 - 1}{4}$, y sea $x_0 = 0.2$, la iteración de fijo correspondiente es

$$x_0 = 0.2, \quad g(x) = (x^4 - 1)/4$$

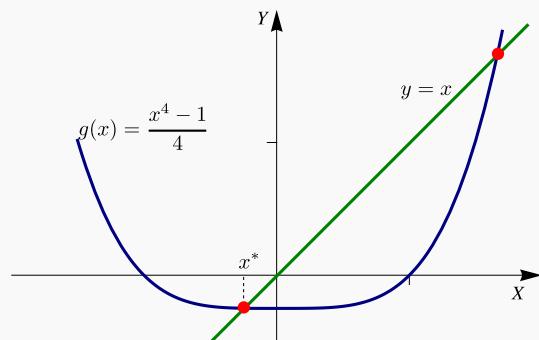
$$x_1 = g(x_0) \approx -0.2496$$

$$x_2 = g(x_1) \approx -0.2490296\dots$$

$$x_3 = g(x_2) \approx -0.2490385\dots$$

...

Toda la información aparece en la tabla



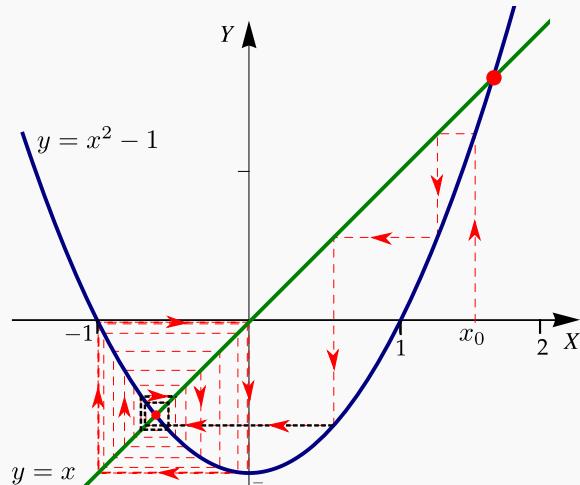
n	x_{n+1}	Error absoluto estimado: $ x_{n+1} - x_n $	Error relativo estimado: $ (x_{n+1} - x_n)/x_{n+1} $
1	-0.2496	0.4496	1.801282051
2	-0.249029672515994	0.000570327	0.002290199
3	-0.249038510826169	8.83831×10^{-6}	3.54897×10^{-5}
4	-0.249038374322083	1.36504×10^{-7}	5.48125×10^{-7}
5	-0.249038376430443	2.10836×10^{-9}	8.46601×10^{-9}
6	-0.249038376397879	3.25645×10^{-11}	1.30761×10^{-10}
7	-0.249038376398382	5.02959×10^{-13}	2.0196×10^{-12}

Ejemplo 4.7

Consideremos el problema de punto fijo $x^2 - 1 = x$. De acuerdo a la figura, hay un punto fijo en $[1, 2]$ y otro en $[-1, 0]$. Estos puntos son $x^* = \frac{1}{2}(1 - \sqrt{5}) \approx -0.618034$ y $x^* = \frac{1}{2}(1 + \sqrt{5})$.

Si tratamos de aproximar estos puntos fijos iniciando en $x_0 = 1.5$, la iteración de punto fijo parece divergente según los datos de la tabla. La iteración cae en un ciclo si para algún n , $x_n = 0$ o $x_n = -1$.

n	x_n	Error estimado
0	1.5	
1	1.25000000	0.25
2	0.56250000	0.6875
3	-0.68359375	1.2460937
...	...	
22	0	1
23	-1	1
24	0	1
25	-1	1
26	0	1



La iteración parece que se acerca al punto $x^* \approx -0.618034$ pero después la iteración se aleja de x^* hasta caer en el ciclo $0, -1, 0, -1, 0, -1, \dots$. En ambos puntos fijos hay un entorno alrededor de ellos en el que $|g'(x)| > 1$.

Ejemplo 4.8

Aunque no es un método eficiente (en esta forma cruda), podemos aproximar la única solución real de $x^3 + x + 1 = 0$ usando iteración de punto fijo.

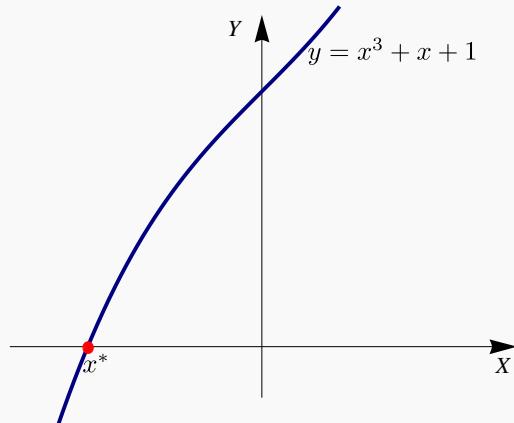
Para empezar, debemos poner el problema como un problema de punto fijo. Para hacer esto debemos despejar “ x ”. Hay varias posibilidades.

- a.) $x = -1 - x^3,$
- b.) $x = \sqrt[3]{-1 - x},$
- c.) $x = \frac{-1 - x}{x^2},$
- d.) ...

El resultado de aplicar punto fijo en los dos primeros casos, con $x_0 = -0.5$, se puede ver en las tablas que siguen.

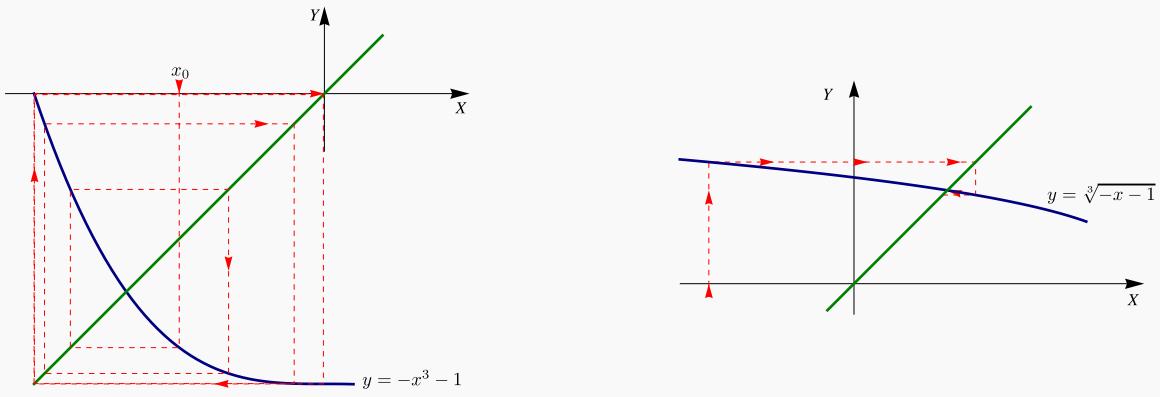
n	$x = -1 - x^3$
1	-0.7840000000
2	-0.5181096960
3	-0.8609198471
4	-0.3619008595
5	-0.9526010366
...	...
17	-1
18	0
19	-1
20	0
21	-1
22	0
...	

Tabla 4.1: Punto fijo aplicado a $x = -1 - x^3$.



n	$x = \sqrt[3]{-x - 1}$	$ x_n - x_{n+1} $
1	-0.79370052	0.293700526
2	-0.59088011	0.202820413
3	-0.74236393	0.151483819
4	-0.63631020	0.106053729
...		
24	-0.6822715	0.000134782
25	-0.6823680	9.65013×10^{-5}
26	-0.6822989	6.90905×10^{-5}
27	-0.6823484	4.94671×10^{-5}
28	-0.6823130	3.54165×10^{-5}
29	-0.6823383	2.53572×10^{-5}
30	-0.6823202	1.81548×10^{-5}

Tabla 4.2: Punto fijo aplicado a $x = \sqrt[3]{-x - 1}$.



Los teoremas que siguen dan condiciones *suficientes* (pero no necesarias) para que haya un punto fijo en un intervalo, condiciones suficientes para que el punto fijo sea único en el intervalo y condiciones suficientes para la convergencia.

Teorema 4.2

Si $g \in C[a, b]$ y si $g(x) \in [a, b]$ para todo $x \in [a, b]$, entonces g tiene un punto fijo en $[a, b]$

Teorema 4.3

Si $g \in C[a, b]$ y si $g(x) \in [a, b]$ para todo $x \in [a, b]$ y si g' esta definida en $]a, b[$ y cumple $|g'(x)| < 1$ en este intervalo, entonces el punto fijo es único.

Teorema 4.4

Si $g \in C[a, b]$ y si $g(x) \in [a, b]$ para todo $x \in [a, b]$ y si g' esta definida en $]a, b[$ y existe k positiva tal que $|g'(x)| \leq k < 1$ en $]a, b[$, entonces, para cualquier $x_0 \in [a, b]$, la iteración $x_{n+1} = g(x_n)$ converge a un único punto fijo x^* de g en este intervalo. También

$$|x_{n+1} - x^*| \leq \frac{k^n |x_1 - x_0|}{1 - k}, \quad n = 1, 2, \dots$$

El orden de convergencia de este método coincide con la multiplicidad del punto fijo, es decir, si $g^{(p)}$ es la primera derivada que no se anula en x^* entonces el orden de convergencia es p .

Ejemplo 4.9

Sea $g(x) = (x^3 - 1)/4$. En el intervalo $[-1, 1]$, esta función tiene un punto fijo único x^* y además $\lim_{n \rightarrow \infty} g(x_n) = x^*$.

Para mostrar esta afirmación verifiquemos las condiciones del teorema (4.4).

- g es continua y $g(x) \in [-1, 1]$ para todo $x \in [-1, 1]$

Efectivamente, g es continua y como $g'(x) = \frac{3x^2}{4}$ entonces g solo tiene un punto crítico en $[-1, 1]$, $x = 0$. Luego,

$$\text{Mín}\{g(-1), g(1), g(0)\} \leq g(x) \leq \text{Máx}\{g(-1), g(1), g(0)\}$$

$$\text{Mín}\{-\frac{1}{2}, 0, -\frac{1}{4}\} \leq g(x) \leq \text{Máx}\{-\frac{1}{2}, 0, -\frac{1}{4}\} \implies -1 \leq -\frac{1}{2} \leq g(x) \leq 0 \leq 1$$

por lo tanto, $g(x) \in [-1, 1]$. Con lo cual g tiene un punto fijo en $[-1, 1]$.

- Si g' existe en $] -1, 1[$ y $|g'(x)| \leq k < 1$ para toda $x \in] -1, 1[$ entonces el punto fijo es único y la iteración de punto fijo converge para cualquier $x_0 \in [-1, 1]$.

Como $g'(x) = \frac{3x^2}{4}$ entonces $g''(x) = \frac{3x}{2}$ tiene un punto crítico en el intervalo, a saber $x = 0$. Por lo tanto

$$|g'(x)| \leq \text{Máx}\{|g'(-1)|, |g'(1)|, |g(0)|\} \implies |g'(x)| \leq \frac{3}{4} < 1 \text{ si } x \in [-1, 1]$$

por lo que, en particular

$$|g'(x)| \leq \frac{3}{4} < 1 \text{ en }] -1, 1[$$

y podemos tomar $k = 3/4$. Por lo tanto, para cualquier $x_0 \in [-1, 1]$, $\lim_{n \rightarrow \infty} g(x_n) = x^*$.

Además, si $x_0 = -0.5$, una cota de error en la n -ésima iteración de punto fijo se puede establecer como

$$|x_n - x^*| \leq \frac{(3/4)^n}{1 - 3/4} |-0.5 - g(-0.5)| \implies |x_n - x^*| \leq 3^n 4^{1-n} \frac{7}{32}$$

En particular, cuando $n = 5$, $x_5 = -0.2541018552\dots$ y entonces $|x_5 - x^*| \leq 0.2076416015625$, mientras que la estimación del valor absoluto, más apagada a la realidad en este caso, nos da $3,27797 \times 10^{-6}$

Criterio de parada.

Vamos a implementar una función `puntofijo(g, x0, tol, maxIteraciones)`. Para la implementación del método de punto fijo, el criterio de parada que podríamos usar es

$$|x_n - x_{n-1}| \leq \text{tol} \quad \text{y un número máximo de iteraciones.}$$

Puesto que $x^* = g(x^*)$ y como $x_{k+1} = g(x_k)$, entonces, usando el teorema del valor medio para derivadas, obtenemos

$$x^* - x_{k+1} = g(x^*) - g(x_k) = g'(\xi_k)(x^* - x_k) \quad \text{con } \xi_k \text{ entre } x^* \text{ y } x_k$$

Ahora, como $x^* - x_k = (x^* - x_{k+1}) + (x_{k+1} - x_k)$ se obtiene que

$$x^* - x_k = \frac{1}{1 - g'(\xi_k)}(x_{k+1} - x_k)$$

Por tanto, si $g'(x) \approx 0$ en un entorno alrededor de x^* , la diferencia $|x_{k+1} - x_k|$ sería un estimador del error. Caso contrario si g' se acerca a 1.

■ **Implementación con R.** Vamos a implementar una función `puntofijo(g, x0, tol, maxIteraciones)`, el criterio de parada que podríamos usar es

$$|x_n - x_{n-1}| \leq \text{tol} \quad \text{y un número máximo de iteraciones.}$$

Algoritmo 4.1: Iteración de Punto fijo.

Datos: Una función continua g , x_0 , tol , maxItr .

Salida: Si hay convergencia, una aproximación x_n de un punto fijo.

```

1 k = 0;
2 repeat
3   | x1 = g(x0);
4   | dx = |x1 - x0|;
5   | x0 = x1;
6   | k = k + 1;
7 until dx ≤ tol o k > maxItr ;
8 return x1

```

■ Código R 4.1: Iteración de “punto fijo”

```
#1e-9 = 0.000000001
puntofijo =function(g, x0, tol=1e-9, maxIteraciones=100){
```

```

k = 1
# iteración hasta que abs(x1 - x0) <= tol o se alcance maxIteraciones
repeat{
  x1 = g(x0)
  dx = abs(x1 - x0)
  x0 = x1
  #Imprimir estado
  cat("x_", k, "= ", x1, "\n")
  k = k+1
  #until
  if(dx< tol|| k > maxIteraciones) break;
}
# Mensaje de salida
if( dx > tol ){
  cat("No hubo convergencia   ")
  #return(NULL)
} else{
  cat("x* es aproximadamente ", x1, " con error menor que ", tol)
}
}

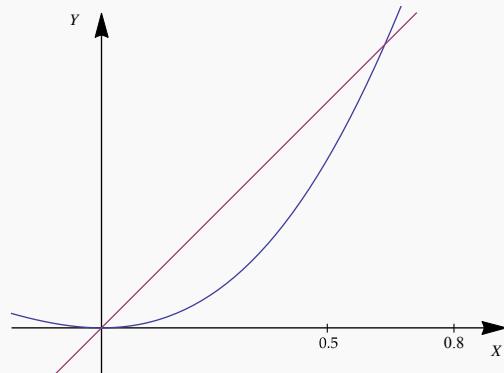
```

Ejemplo 4.10

La ecuación $(x+1)\sin(x^2) - x = 0$ tiene dos soluciones en $[0, 1]$. Una manera de expresar el problema de encontrar una solución de esta ecuación en términos de un problema de punto fijo es escribir la ecuación como

$$(x+1)\sin(x^2) = x$$

De acuerdo a la gráfica, hay dos puntos fijos, y uno de ellos x_1^* , está en $[0.5, 0.8]$. Por la forma de la gráfica, parece que el método de punto fijo no va a detectar a x_1^* .



Corremos nuestro programa con valores $x0 = 0.8$ y $x0= 0.5$ y observamos que, en ambos casos, el método solo converge a $x_0^* = 0$.

```

g = function(x) (x+1)-sin(x^2)
puntofijo(g, 0.5, 1e-9) # maxIteraciones=100

```

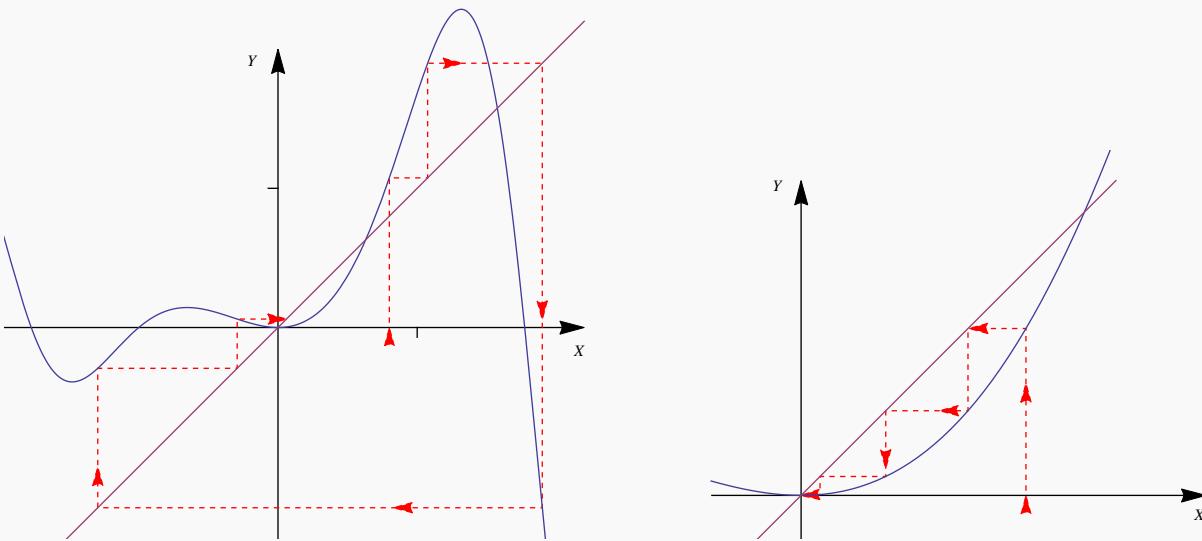
```

# x_ 1 =  0.3711059
# x_ 2 =  0.1882318
# x_ 3 =  0.0420917
# x_ 4 =  0.001846285
# x_ 5 =  3.415062e-06
# x_ 6 =  1.166269e-11
# x_ 7 =  1.360183e-22
# x* es aproximadamente 1.360183e-22 con error menor que 1e-09

puntofijo(g, 0.8, 1e-9)
#x_ 1 =  1.074952
#x_ 2 =  1.898592
#x_ 3 =  -1.294765
#x_ 4 =  -0.2931222
#x_ 5 =  0.06066068
#x_ 6 =  0.003902924
#x_ 7 =  1.529227e-05
#x_ 8 =  2.33857e-10
#x_ 9 =  5.468908e-20
#x* es aproximadamente 5.468908e-20 con error menor que 1e-09

```

La representación gráfica del método nos muestra con más detalle qué puede estar pasando.



3

Ejercicios

4.1 Usar punto fijo para aproximar la solución de cada ecuación en el intervalo que se indica.

- a.) $x^3 - x - 1 = 0$ en $[1, 2]$
- b.) $x^2 - x - 1 = 0$ en $[-1, 0]$
- c.) $x = e^{-x}$ en $[0, 1]$
- d.) $x^5 - x + 1 = 0$ en $[-2, -1]$
- e.) $\sin(x) - x = 0$ en $[-1, 1]$

4.2 Verifique que las siguientes funciones tienen un único punto fijo en el intervalo dado.

- a.) $g(x) = (x^2 - 1)/3$ en el intervalo $[-1, 1]$.
- b.) $g(x) = 2^{-x}$ en $[1/3, 1]$

4.3 Repita la parte 1.) del ejemplo (4.9) con $g(x) = (x^3 - 1)/4$ en $[-0.5, 0]$.

4.4 Considere $g(x) = 2^{-x}$.

- a.) ¿Podría encontrar una constante positiva $k < 1$ tal que $|g'(x)| \leq k \quad \forall x \in [1/3, 1]$?
- b.) ¿Se puede garantizar que la iteración de punto fijo, iniciando en cualquier $x_0 \in [1/3, 1]$, converge al único punto fijo de g en el intervalo $[1/3, 1]$?

4.5 Considere el problema de punto fijo $x = 0.5(\sin(x) + \cos(x))$. Determine un intervalo $[a, b]$ donde la iteración de punto fijo converge sin importar la elección de la aproximación inicial $x_0 \in [a, b]$. Debe justificar su respuesta.

4.6 Usando la implementación, aplique iteración de punto fijo para resolver el problema $x = (2x^3 - 2)/(3x^2 - 3)$ tomando $x_0 = 1.2$. Hay algo muy extraño pasando aquí. ¿Qué es?

4.7 Considere el problema de punto fijo $x = e^{-x}$. Muestre que la iteración de punto fijo converge para cualquier $x_0 > 0$.

4.8 Considere el problema de punto fijo $x = 3x - 3x^2 + x^3$. Determine un intervalo $[a, b]$ tal que la iteración de punto fijo converja a $x^* = 1$, para todo $x_0 \in [a, b]$. **Ayuda:** Haga una representación gráfica de g , g' y 1.

4.9 [5 puntos] (Dinámica de poblaciones) En el estudio de las poblaciones (por ejemplo de bacterias), la

ecuación $x = xR(x)$ establece un vínculo entre el número de individuos en una generación X y el número de individuos en la siguiente generación. La función $R(x)$ modela la tasa de cambio de la población considerada.

Llamemos $\phi(x) = xR(x)$. La dinámica de una población se define por el proceso iterativo

$$x_k = \phi(x_{k-1}) \text{ con } k \geq 1,$$

donde x_k representa el número de individuos presentes k generaciones más tarde de la generación x_0 inicial. Por otra parte, los estados estacionarios (o de equilibrio) x^* de la población considerada son las soluciones de un problema de punto fijo

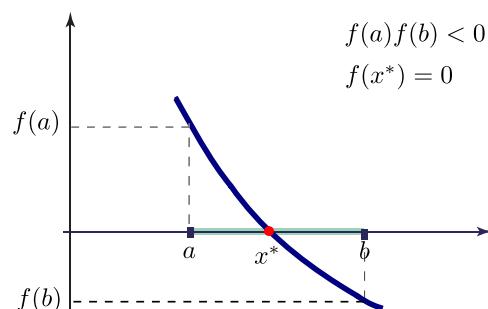
$$x^* = \phi(x^*),$$

Vamos a usar el modelo “predador-presa con saturación”: $R(x) = \frac{rx}{1 + (x/K)^2}$ con $r = 3$ y $K = 1$

- Graficar la función $\phi(x) = xR(x)$ con $x \in [0, 5]$, junto con la función $y = x$ (para la función $y = x$ usar `curve(1*x, ...)`). Además usar `abline()` para los ejes.
- De acuerdo a la gráfica, elija una valor inicial x_0 cerca del punto fijo donde la función ϕ se empieza a estabilizar y obtenga la aproximación a este punto fijo usando el programa que ya implementamos.

4.3 El método de Bisección

Este es uno de los métodos más sencillos y de fácil intuición, para resolver ecuaciones en una variable. Se basa en el Teorema de los Valores Intermedios, el cual establece que toda función continua f en un intervalo cerrado $[a, b]$ ($f \in C[a, b]$) toma todos los valores que se hallan entre $f(a)$ y $f(b)$. Esto es, que todo valor entre $f(a)$ y $f(b)$ es la imagen de al menos un valor en el intervalo $[a, b]$.

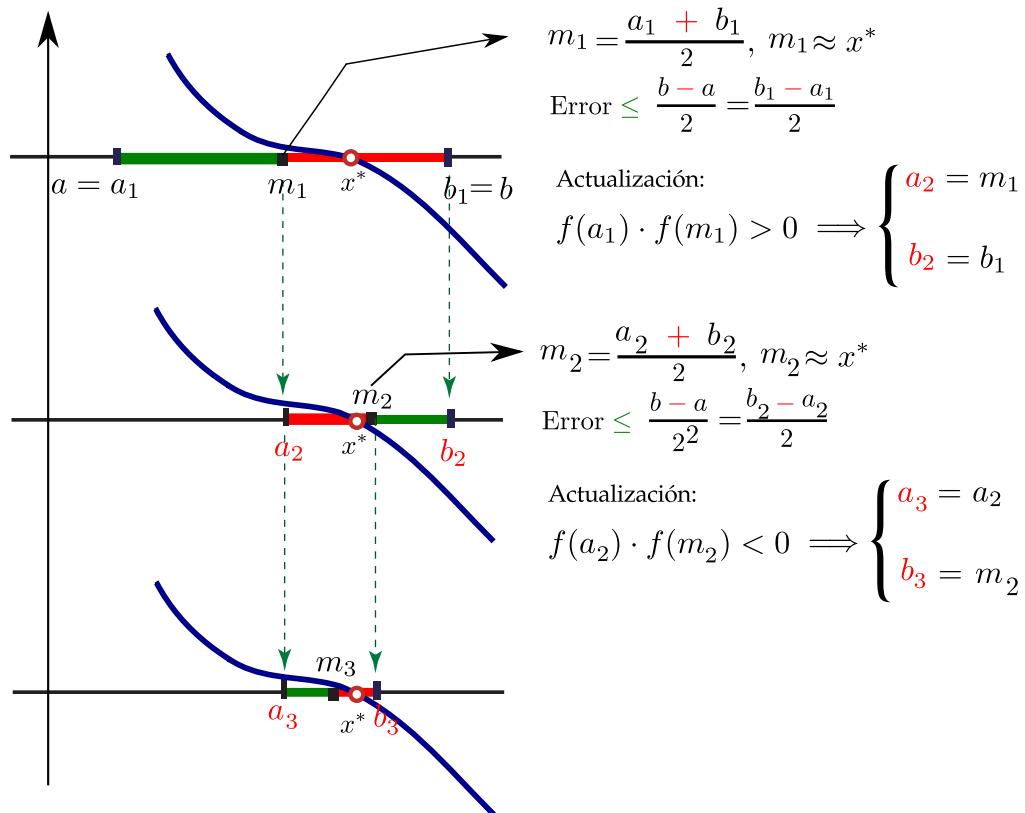


En caso de que $f(a)$ y $f(b)$ tengan signos opuestos (es decir, $f(a) \cdot f(b) < 0$), el valor cero sería un valor intermedio entre $f(a)$ y $f(b)$, por lo que con certeza existe un x^* en $[a, b]$ que cumple $f(x^*) = 0$. De esta forma, se asegura la

existencia de al menos una solución de la ecuación $f(x) = 0$.

El método consiste en lo siguiente: Supongamos que en el intervalo $[a, b]$ hay un cero de f . Calculamos el punto medio $m = (a + b)/2$ del intervalo $[a, b]$. A continuación calculamos $f(m)$. En caso de que $f(m)$ sea igual a cero, ya hemos encontrado la solución buscada. En caso de que no lo sea, verificamos si $f(m)$ tiene signo opuesto al de $f(a)$. Se redefine el intervalo $[a, b]$ como $[a, m]$ o $[m, b]$ según se haya determinado en cuál de estos intervalos ocurre un cambio de signo. A este nuevo intervalo se le aplica el mismo procedimiento y así, sucesivamente, iremos encerrando la solución en un intervalo cada vez más pequeño, hasta alcanzar la precisión deseada.

En la siguiente figura se ilustra el procedimiento descrito.



El procedimiento construye tres sucesiones a_n , b_n y m_n ,

- Para $k = 1, 2, \dots$, $m_k = \frac{b_k + a_k}{2}$ y $[a_k, b_k] = \begin{cases} [m_{k-1}, b_{k-1}] & \text{si } f(a_{k-1})f(m_{k-1}) > 0 \\ [a_{k-1}, m_{k-1}] & \text{si } f(a_{k-1})f(m_{k-1}) < 0 \end{cases}$

- **Estimación del error:** El error exacto en el k -ésimo paso es $|m_k - x^*|$. Geométricamente se puede ver que esto es menos que la mitad del intervalo $[a_k, b_k]$, es decir

$$|m_k - x^*| \leq \frac{b_k - a_k}{2} = \frac{b - a}{2^k}$$

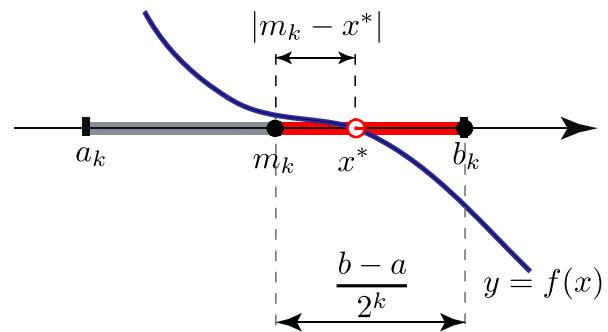


Figura 4.3: Estimación del error en bisección.

Ejemplo 4.11

Aplicar el método de bisección para aproximar la solución de la ecuación $x^2 = \cos(x) + 1$ en $[1, 2]$

- Debemos reescribir la ecuación como $x^2 - \cos(x) - 1 = 0$. En este caso, $f(x) = x^2 - \cos(x) - 1$. Podemos hacer una gráfica de esta función y observar que esta función tiene un cero en el intervalo $[1, 2]$ pues, efectivamente $f(1) \cdot f(2) = -1.84575... < 0$.

Calculemos ahora a_k , b_k y m_k así como la estimación del error.

$$k = 1 : \quad a_1 = 1, \quad b_1 = 2 \text{ y } m_1 = \frac{a_1 + b_1}{2} = 1.5. \quad \text{Error} \leq 0.5$$

$$k = 2 : \quad f(a_1) \cdot f(m_1) = -0.637158 < 0,$$

$$a_2 = 1, \quad b_2 = 1.5 \text{ y } m_2 = \frac{a_2 + b_2}{2} = 1.25. \quad \text{Error} \leq 0.25$$

$$k = 3 : \quad f(a_2) \cdot f(m_2) = -0.13355064 < 0,$$

$$a_3 = 1, \quad b_3 = 1.25 \text{ y } m_3 = \frac{a_3 + b_3}{2} = 1.125. \quad \text{Error} \leq 0.125$$

$$\vdots \quad \vdots$$

$$k = 44 : \quad m_{44} = 1.17650193990184. \quad \text{Error} \leq 2.84 \times 10^{-14}$$

- Así, $m_{44} = 1.17650193990184$ aproxima el cero de $f(x) = x^2 - \cos(x) - 1$ en $[1, 2]$ con un error $\leq 2.84 \times 10^{-14}$

Podemos poner estos cálculos (junto con otros adicionales) en una tabla

k	a_k	b_k	m_k	Error Estimado
1	1	2	1.5	0.5
2	1	1.5	1.25	0.25
3	1	1.25	1.125	0.125
4	1.125	1.25	1.1875	0.0625
...	...			
44	1.176501939	1.176501939	1.176501939	2.84217×10^{-14}

4.4 Algoritmo e Implementación.

En la implementación del método de bisección, en vez de usar $f(a)f(m) < 0$ ponemos $\text{sgn}(f(a)) \neq \text{sgn}(f(m))$, de esta manera nos ganamos una multiplicación (que es claramente innecesaria).

En bisección es mejor calcular m como $m = a + (b - a)/2$. Con esto somos consistentes con la estrategia general (en análisis numérico) de calcular una cantidad agregando una corrección a la aproximación anterior. Además ganamos algo en precisión.

El criterio de parada es: Detenerse en m_k si $\frac{b_k - a_k}{2} = \frac{b - a}{2^k} \leq \delta$

Figura 4.4: Algoritmo de Bisección.

Datos: a, b, tol y f continua en $[a,b]$ con $f(a)f(b) < 0$.

Salida: Una aproximación m de un cero x^* de f en $[a, b]$.

```

1  $k = 0;$ 
2 repeat
3    $m = a + 0.5(b - a);$ 
4   if  $f(m) = 0$  then
5      $\quad$  return  $m$ ; break
6    $dx = (b - a)/2;$ 
7   if  $\text{Sgn}(f(a)) \neq \text{Sgn}(f(m))$  then
8      $\quad$   $b = m;$ 
9   else
10     $\quad$   $a = m$ 
11    $k = k + 1$ 
12 until  $dx \leq \text{tol};$ 
13 return  $m$ 

```

Implementación en R .

En esta implementación usamos **formatC** para formatear de manera limpia el resultado de cada iteración.

■ Código R 4.2: Método de Bisección

```
biseccion = function(f, xa, xb, tol){
  if( sign(f(xa)) == sign(f(xb)) ){ stop("f(xa) y f(xb) tienen el mismo signo") }
  # a = min(xa,xb)
  # b = max(xa,xb)
  a = xa; b = xb
  k = 0
  #Par imprimir estado
  cat("-----\n")
  cat(formatC( c("a","b","m","Error est."), width = -15, format = "f", flag = " "), "\n")
  cat("-----\n")

repeat{
  m = a + 0.5*(b-a)
  if( f(m)==0 ){ cat("Cero de f en [",xa,";",xb,"] es: ", m ) }
  if( sign(f(a)) != sign(f(m)) ){
    b = m
  } else { a = m }
  dx = (b-a)/2
  # imprimir estado
  cat(formatC( c(a,b,m,dx), digits=7, width = -15, format = "f", flag = " "), "\n")
  k = k+1
  #until
  if( dx < tol ){
    cat("-----\n\n")
    cat("Cero de f en [",xa,";",xb,"] es approx: ", m, "con error <=", dx)
    break;
  }
} #repeat
}

## Pruebas
f = function(x) x-cos(x)
curve(f, -2,2); abline(h=0, v=0) #gráfico para decidir un intervalo
biseccion(f, 0.5, 0.8, 0.000001)

# -----
#   a           b           m           Error est.
# -----
# 0.6500000  0.8000000  0.6500000  0.0750000
# 0.7250000  0.8000000  0.7250000  0.0375000
# 0.7250000  0.7625000  0.7625000  0.0187500
# 0.7250000  0.7437500  0.7437500  0.0093750
```

```

# 0.7343750    0.7437500    0.7343750    0.0046875
# 0.7390625    0.7437500    0.7390625    0.0023437
# 0.7390625    0.7414063    0.7414063    0.0011719
# 0.7390625    0.7402344    0.7402344    0.0005859
# 0.7390625    0.7396484    0.7396484    0.0002930
# 0.7390625    0.7393555    0.7393555    0.0001465
# 0.7390625    0.7392090    0.7392090    0.0000732
# 0.7390625    0.7391357    0.7391357    0.0000366
# 0.7390625    0.7390991    0.7390991    0.0000183
# 0.7390808    0.7390991    0.7390808    0.0000092
# 0.7390808    0.7390900    0.7390900    0.0000046
# 0.7390808    0.7390854    0.7390854    0.0000023
# 0.7390831    0.7390854    0.7390831    0.0000011
# 0.7390842    0.7390854    0.7390842    0.0000006
# -----
#
# # Cero de f en [ 0.5 , 0.8 ] es approx: 0.739084243774414 con error <= 5.72204589821546e-07

```

La implememntación del paquete **pracma** no usa como criterio de parada una tolerancia, más bien hace bisección de los intervalos hasta que el punto medio coincide con alguno de los extremos del intervalo actual (o que se completen 100 iteraciones).

■ Código R 4.3:**bisect**: implememntación del paquete “pracma”

```

bisect <- function(f, a, b, maxiter = 100, tol = NA)
# Bisection search, trimmed for exactness, not no. of iterations
{
  if (!is.na(tol)) warning("Deprecated: Argument 'tol' not used anymore.")
  if (f(a)*f(b) > 0) stop("f(a) and f(b) must have different signs.")
  x1 <- min(a, b); x2 <- max(a,b)
  xm <- (x1+x2)/2.0
  n <- 1
  while (x1 < xm && xm < x2 && n < maxiter) {
    n <- n+1
    if (sign(x1) != sign(x2) && x1 != 0 && x2 != 0) {
      xm <- 0.0
      if (f(xm) == 0.0) {x1 <- x2 <- xm; break}
    }
    if (sign(f(x1)) != sign(f(xm))) {
      x2 <- xm
    } else {
      x1 <- xm
    }
    xm <- (x1 + x2) / 2.0
  }
}
```

```

}

return(list(root=xm, f.root=f(xm), iter=n, estim.prec=abs(x1-x2)))
}

regulaFalsi <- function(f, a, b, maxiter = 100, tol = .Machine$double.eps^0.5)
#Regula Falsi search for zero of a univariate function in a bounded interval
{
  x1 <- a;      x2 <- b
  f1 <- f(x1); f2 <- f(x2)
  if (f1*f2 > 0) stop("f(a) and f(b) must have different signs.")

  m <- 0.5                                # Illinois rule
  niter <- 0
  while (abs(x2-x1) >= tol && niter <= maxiter) {
    niter <- niter + 1
    x3 <- (x1*f2-x2*f1)/(f2-f1); f3 <- f(x3)

    if(f3*f2 < 0) {
      x1 <- x2;  f1 <- f2
      x2 <- x3;  f2 <- f3
    } else {
      # m <- f2/(f2+f3)                      # Pegasus rule
      # m <- if (1-f3/f2 > 0) 1-f3/f2 else 0.5 # Andersen/Bjoerk
      f1 <- m * f1
      x2 <- x3;  f2 <- f3
    }
  }
  if (niter > maxiter && abs(x2-x1) >= tol)
    cat("regulaFalsi stopped without converging.\n")
  return(list(root = x3, f.root = f3, niter = niter, estim.prec = x1-x2))
}

```

Podemos usar el método **bisect()** del paquete **pracma** de la siguiente manera,

■ Código R 4.4:**bisect()** (paquete “**pracma**”)

```
#install.packages(pracma)
require(pracma)

f = function(x) x-cos(x)
bisect(f, 0.5, 0.8, 35)

#$root
#[1] 0.739085133207846
```

```

##$f.root
#[1] -1.22426513371465e-11

##$iter
#[1] 35

##$estim.prec
#[1] 1.74622538651192e-11

```

4.5 Orden de convergencia y número de iteraciones.

Sea x^* es el único cero de f en $[a, b]$. En la k -ésima iteración, al aproximar x^* con m_k se tiene que

$$|m_k - x^*| \leq \frac{b-a}{2^k}, \quad k = 1, 2, \dots$$

entonces,

- Si tenemos una *tolerancia* $\delta > 0$, y si queremos cortar la sucesión m_n en la k -ésima iteración m_k de tal manera que $|m_k - x^*| \leq \delta$ entonces podemos estimar el número k de iteraciones con

$$|m_k - x^*| \leq \frac{b-a}{2^k} \leq \delta$$

Tomando logaritmo natural a ambos lados de $\frac{b-a}{2^k} \leq \delta$ obtenemos

$$k \geq \ln\left(\frac{b-a}{\delta}\right) / \ln 2 \text{ iteraciones}$$

excepto que en algún momento $f(m_j) = 0$ para algún $j < k$

De aquí podemos deducir que si $|b-a| < 1$ y si $\delta = 2^{-52}$ entonces el número de iteraciones necesarias para alcanzar esta tolerancia δ es como mínimo $k = 52$.

- Observe que $2^{-3.4} \approx 10^{-1}$ y en general $2^{-3.4 \cdot d} \leq 10^{-d}$. Como $|m_k - x^*| \leq \frac{b-a}{2^k}$, esto nos dice que si $b-a \leq 1$ entonces bisección se gana un dígito decimal cada 3.4 iteraciones aproximadamente.

Ejemplo 4.12

Al aplicar el algoritmo de bisección a una función continua f con un único cero en el intervalo $[-2, 1]$, si queremos que el error de aproximación sea $\leq 0.00005 = 0.5 \times 10^{-4}$, el número k de iteraciones debe cumplir

$$k \geq \ln\left(\frac{3 \cdot 10^4}{0.5}\right) / \ln 2 = 15.873$$

por lo que deben realizarse por lo menos 16 iteraciones. Verifíquelo!

Notas.

- a.) El método de bisección, la única información que usa es el signo de f . El número de iteraciones no depende, en general, de la función f , depende del intervalo y la tolerancia δ . La única manera de que el número de iteraciones sea menor es que se dé el caso $f(m_j) = 0$, para algún $j < k$.
- b.) Aunque el método de bisección es *robusto* al nivel de la precisión de la máquina, puede pasar que la aproximación *no* quede numéricamente bien determinada si la función es muy “aplanada”.

4.6 El Método de Newton

El método de Newton (llamado a veces método de Newton-Raphson¹) es uno de los métodos que muestra mejor velocidad de convergencia llegando (bajo ciertas condiciones) a duplicar, en cada iteración, los decimales exactos.

Si f es una función tal que f , f' y f'' existen y son continuas en un intervalo I y si un cero x^* de f está en I , se puede construir una sucesión $\{x_n\}$ de aproximaciones, que converge a x^* (bajo ciertas condiciones) de la manera que se describe a continuación: Si x_0 está *suficientemente cercano* al cero x^* , entonces supongamos que h es la *corrección* que necesita x_0 para alcanzar a x^* , es decir, $x_0 + h = x^*$ y $f(x_0 + h) = 0$. Como

$$0 = f(x_0 + h) \approx f(x_0) + h f'(x_0)$$

entonces ‘despejamos’ *la corrección* h ,

$$h \approx -\frac{f(x_0)}{f'(x_0)}$$

De esta manera, una aproximación *corregida* de x_0 sería

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$



¹ En la literatura inglesa se acostumbra llamar al método de Newton, método de Newton-Raphson. El método de Newton aparece en su libro ‘Method of Fluxions’ escrito en 1671 pero publicado hasta 1736. El método fue publicado por primera vez en un libro de J. Raphson en 1690. Se dice que Raphson tuvo acceso al manuscrito de Newton. En todo caso ni Newton ni Raphson mencionan las derivadas. El primero que dio una descripción del método de Newton usando derivadas fue T. Simpson en 1740.

Aplicando el mismo razonamiento a x_1 obtendríamos la aproximación $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$ y así sucesivamente.

■ **Geométricamente** se vería así: Partiendo de una aproximación x_0 de un cero x^* de f , entonces x_1 es la intersección, de la recta tangente a f en x_0 , con el eje X . Cuando se ha calculado una aproximación x_n , la siguiente aproximación x_{n+1} se obtiene hallando la intersección con el eje X de la recta tangente en el punto $(x_n, f(x_n))$. El proceso se muestra en las figuras (4.5), (4.6) y (4.7).

La intersección de la tangente con el eje X se obtiene resolviendo $y = 0$. Como la ecuación de una recta de pendiente m que pasa por (x_0, y_0) es $y = m(x - x_0) + y_0$ entonces la ecuación de la recta tangente en $(x_n, f(x_n))$ es

$$y = f'(x_n)(x - x_n) + y_n$$

Al despejar el valor de x , se obtiene x_{n+1} :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

En general, en la fórmula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, $f(x_n)$ no presenta grandes variaciones mientras que $f'(x_n)$ puede variar fuertemente: Si $f'(x_n)$ es grande entonces la corrección $\frac{f(x_n)}{f'(x_n)}$ que se le aplica a x_n para aproximar el cero es pequeña. Por otro lado, si $f'(x_n)$ es pequeña entonces la corrección sería mayor por lo que aproximar un cero de f puede ser un proceso lento o a veces imposible.

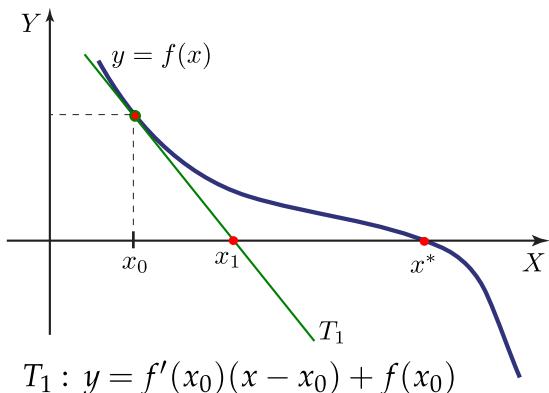


Figura 4.5: Obteniendo x_1

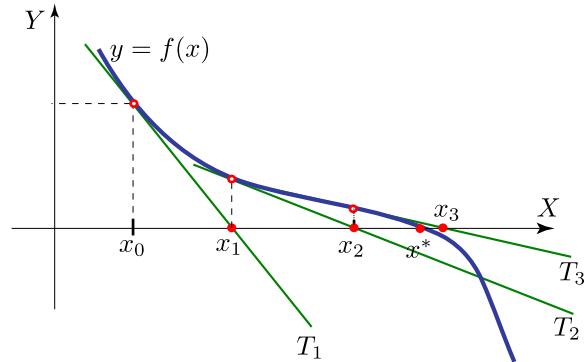


Figura 4.6: Obteniendo x_1, x_2, \dots

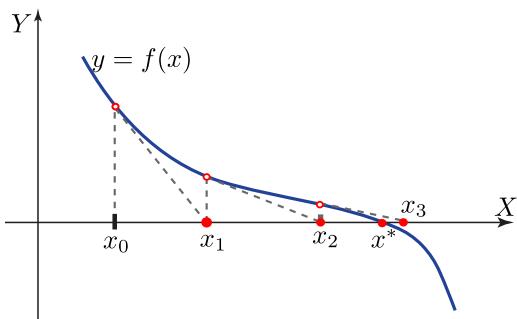


Figura 4.7: Representación simplificada

Ejemplo 4.13

La ecuación $x^2 - \cos(x) - 1 = 0$ tiene una solución x^* en $[1, 2]$. Debemos tomar una aproximación inicial, digamos $x_0 = 1.5$.

$f(x) = x^2 - \cos(x) - 1$ y $f'(x) = 2x + \sin(x)$. Entonces *el esquema iterativo* es

$$x_{n+1} = x_n - \frac{x_n^2 - \cos(x_n) - 1}{2x_n + \sin(x_n)}$$

$$n = 0 : x_0 = 1.5$$

$$n = 1 : x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 1.5 - \frac{f(1.5)}{f'(1.5)} = 1.204999.$$

$$\text{Error} \leq |x_1 - x_0| = 0,295$$

$$n = 2 : x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 1.204999 - \frac{f(1.204999)}{f'(1.204999)} = 1.176789.$$

$$\text{Error} \leq |x_2 - x_1| = 0,0282$$

$$n = 3 : x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 1.176789 - \frac{f(1.176789)}{f'(1.176789)} = 1.1765019.$$

$$\text{Error} \leq |x_3 - x_2| = 0,000287$$

Podemos tabular esta información en una tabla agregando más iteraciones. Si usamos Excel obtenemos

x_{n+1}	$ x_{n+1} - x_n $
1.20499955540054	0.295000445
1.17678931926590	0.028210236
1.17650196994274	0.000287349
1.17650193990183	3.004×10^{-8}
1.17650193990183	4.440×10^{-16}

Compare con la solución $x^* = 1.17650193990183$ (exacta en sus catorce decimales) y observe como, aproximadamente, se *duplican* los decimales correctos desde la segunda iteración.

Algunas veces el método de Newton no converge. En las gráficas de la figura que sigue se muestran dos situaciones donde no hay convergencia.

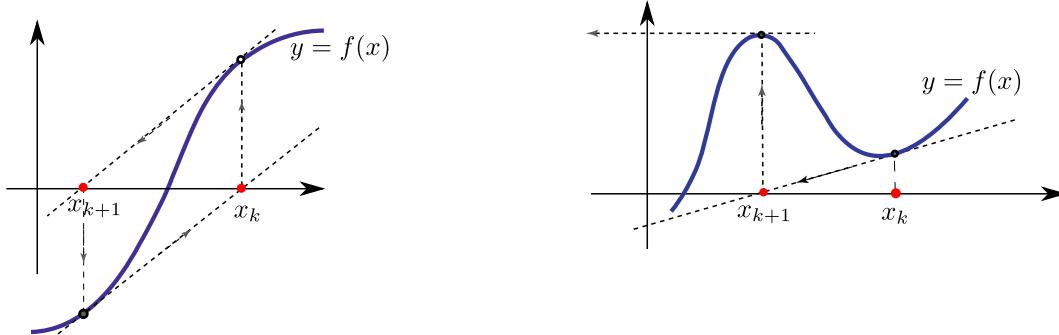
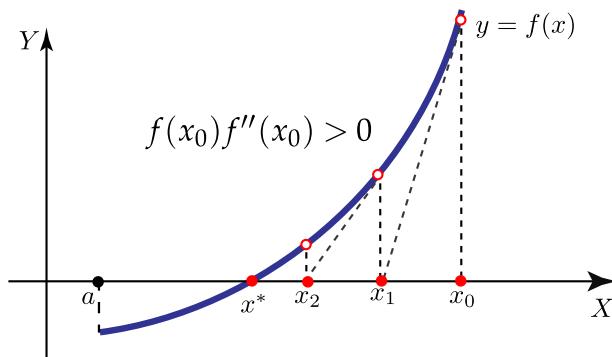


Figura 4.8: Situaciones donde el método de Newton no converge

■ **Convergencia global.** La convergencia del método de Newton *es local*, es decir, como aproximación inicial debe elegir un x_0 que esté “suficientemente cercano” a x^* . Sin embargo hay teoremas que dan criterios para la convergencia global (para ciertos tipos de funciones). Por ejemplo, si en el intervalo $I = [a, b]$ f tiene un único cero y si f' y f'' conservan el signo, entonces una buena aproximación inicial es cualquier $x_0 \in [a, b]$ para el cual $f(x_0)f''(x_0) > 0$ (ver figura).



En el ejemplo (4.13), $x_0 = 1.5$ fue una buena aproximación pues x_0 está en $[1, 2]$ y en este intervalo f' y f'' son positivas y además $f(1.5)f''(1.5) > 0$.

Ejemplo 4.14

Uno de los peligros del método de Newton es *no tomar* una aproximación inicial x_0 suficientemente cercana a x^* . Para ver esto consideremos la ecuación $x^{20} - 1 = 0$, $x > 0$. Esta ecuación tiene una única solución $x^* = 1$. El esquema iterativo es

$$x_{n+1} = \frac{19}{20}x_n + \frac{1}{20x_n^{19}}$$

Si tomamos la aproximación inicial $x_0 = 0.5$ entonces

x_n
$x_1 = 26214.9$
$x_2 = 24904.1$
$x_3 = 23658.9$
$x_4 = 22476.$
$x_5 = 21352.2$
...
$x_{200} = 1.01028$

De hecho, si n es grande entonces $x_{n+1} \approx \frac{19}{20}x_n$, es decir, la corrección es algo pequeña (casi deja igual a x_{n+1}) y toma unas 200 iteraciones llegar cerca de la raíz $x^* = 1$.

Si tomamos la aproximación inicial $x_0 = 0.92$ entonces las cosas cambian dramaticamente

x_n
$x_1 = 1.11778$
$x_2 = 1.06792$
$x_3 = 1.02887$
$x_4 = 1.00654$
$x_5 = 1.00039$
$x_6 = 1.0000014308157694$
$x_7 = 1.0000000000194484$

Ejemplo 4.15 (Ciclos)

Consideremos la ecuación $\sin(x) = 0$, $|x| < \frac{\pi}{2}$. Esta ecuación solo tiene la solución $x^* = 0$. El esquema iterativo es

$$x_{n+1} = x_n - \tan(x_n)$$

Si tomamos x_0 tal que $\tan(x_0) = 2x_0$ entonces

$$x_1 = x_0 - \tan(x_0) = -x_0$$

$$x_2 = -x_0 - \tan(-x_0) = x_0$$

es decir, entramos en un ciclo (figura 4.9). Observe que

$$\tan(x_0) = 2x_0 \implies x_0 = 1.16556\dots$$

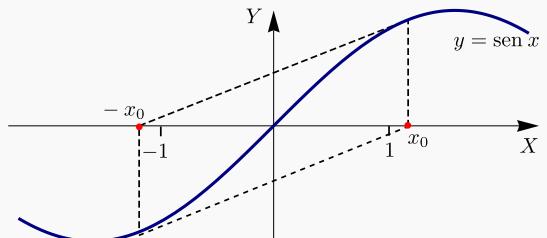


Figura 4.9: Ciclo

Ejemplo 4.16 (Sucesión divergente).

Si a la ecuación $xe^{-x} = 0$ le aplicamos el método de Newton con $x_0 = 2$, obtenemos una sucesión divergente. La sucesión se aleja rápidamente de la raíz $x^* = 0$. En cambio, si ponemos $x_0 = -0.4$, obtenemos una rápida convergencia.

$x_0 = 2$	Error estimado	$x_0 = -0.4$	Error estimado
x_n		x_n	
4	2	-0.114285714	0.285714286
5.333333333	1.333333333	-0.011721612	0.102564103
6.564102564	1.230769231	-0.000135804	0.011585807
7.743826066	1.179723502	-1.84403×10^{-8}	0.000135786
8.892109843	1.148283777	-3.40045×10^{-16}	1.84403×10^{-8}
10.01881867	1.126708829	-1.47911×10^{-31}	3.40045×10^{-16}

Criterio de parada.

El método de Newton se puede ver como un problema de iteración de punto fijo si tomamos

$$g(x) = x - \frac{f(x)}{f'(x)}$$

de esta manera $x_{k+1} = g(x_k)$ se convierte en $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$.

De acuerdo a la subsección 4.2, si $g'(x) \approx 0$ en un entorno alrededor de x^* , la diferencia $|x_{k+1} - x_k|$ sería un buen estimador del error. No sería bueno si g' se acerca a 1.

4.7 Método de Newton: Algoritmo e Implementación.

Figura 4.10: Método de Newton

Datos: $f \in C^2[a, b]$, x_0 , **tol** y **maxItr**.
Salida: Si la iteración converge, una aproximación x_n de un cero de f en $[a, b]$ y una estimación del error.

```

1  k = 0 ;
2  x_k = x_0;
3  repeat
4      dx =  $\frac{f(x_k)}{f'(x_k)}$ ;
5      x_{k+1} = x_k - dx;
6      x_k = x_{k+1};
7      k = k + 1;
8  until dx ≤ tol or k ≤ maxItr ;
9  return x_k y dx

```

Implementación.

Una primera implementación recibe a la función f y a su derivada como parámetros (por nombre). Una segunda implementación usa la función **D()** de la base de **R**, para obtener la derivada. En el capítulo que sigue, usamos “derivación numérica” para implementar una versión de propósito general.

■ Código R 4.5: Método Newton–Raphson.

```

newton1 = function(f, fp, x0, tol, maxiter){
  k = 0
  # Imprimir estado
  cat("-----\n")
  cat(formatC( c("x_k"," f(x_k)","Error est."), "width = -20, format = "f", flag = " "), "\n")
  cat("-----\n")

  repeat{
    correccion = f(x0)/fp(x0)
    x1 = x0 - correccion
    dx = abs(x1-x0)
    # Imprimir iteraciones

```

```

cat(formatC( c(x1 ,f(x1), dx), digits=15, width = -15, format = "f", flag = " "), "\n")
x0 = x1
k = k+1
# until
if(dx <= tol || k > maxiter ) break;
}
cat("-----\n")
if(k > maxiter){
  cat("Se alcanzó el máximo número de iteraciones.\n")
  cat("k = ", k, "Estado: x = ", x1, "Error estimado <= ", correccion)
} else {
  cat("k = ", k, " x = ", x1, " f(x) = ", f(x1), " Error estimado <= ", correccion) }
}

## --- Pruebas
f = function(x) x-cos(x)
fp = function(x) 1+sin(x)
options(digits = 15)
newton1(f,fp, 0, 0.0000005, 10)

# -----
#   x_k           f(x_k)      Error est.
# -----
# 1.000000000000000  0.459697694131860  1.000000000000000
# 0.750363867840244  0.018923073822117  0.249636132159756
# 0.739112890911362  0.000046455898991  0.011250976928882
# 0.739085133385284  0.000000000284721  0.000027757526078
# 0.739085133215161  0.000000000000000  0.000000000170123
# -----
#   k = 5 x =  0.739085133215161 f(x) =  0 Error estimado <=  1.70123407014033e-10

## --- Comentario: con 15 decimales:
# x = 0.739085133215161 y f(0.739085133215161)=5.55111512312578e-16

```

En la implementación que sigue, se usa la función `D()` para calcular la derivada de manera simbólica. En este caso, la implementación recibe la función f usando una tira.

■ Código R 4.6: Método Newton–Raphson.

```

newtonraphson = function(fun, x0, tol = 0.000000005, maxiter = 100){
  # f = string
  numiter = 0
  g = parse(text=fun)      # parse devuelve tipo "expression"

```

```

g. = D(g,"x")
fx = function(x){eval(g)} # convertir f a función
fp = function(x){eval(g.)} # convertir f' a función

correccion = -fx(x0)/fp(x0)

while (abs(correccion) >= tol && numiter <= maxiter) {
  numiter = numiter + 1
  if (fp(x0) == 0) stop("División por cero")
  x1 = x0 + correccion
  correccion = -fx(x1)/fp(x1)
  x0 = x1
}
if (numiter > maxiter){ warning("Se alcanzó el máximo número de iteraciones.")
  cat("Estado:\n")
  cat("k = ", k, " x = ", x1, " f(x) = ", f(x1), "Error estimado <= ", correccion)
} else {
  return(list(cero = x0, f.cero = fx(x0), numeroiter=numiter, error.est = correccion))
}
}

## --- Pruebas
# recibe la función como una tira
newtonraphson("x-cos(x)", 0, 0.00000005, 10)

#$cero
#[1] 0.739085133385284

#$f.cero
#[1] 2.84720580445708e-10

#$numeroiter
#[1] 4

#$error.est
#[1] -1.70123407014033e-10

```

En el capítulo que sigue, veremos la opción cambiar la derivada simbólica, por la derivada numérica.

4

4.10 (Raíz cuadrada) Como \sqrt{A} es una solución de la ecuación $x^2 - A = 0$, podemos usar el método de Newton para estimar \sqrt{A} (figura 4.11). Como veremos más adelante, la sucesión converge para cualquier

$$x_0 > 0$$

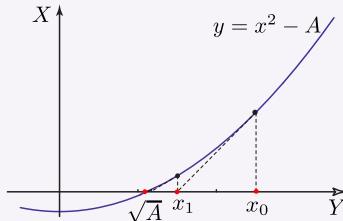


Figura 4.11: La sucesión converge a la raíz para cualquier valor $x_0 > 0$

- a.) Verifique que la fórmula de iteración para obtener la estimación de la raíz cuadrada es $x_n = 0.5\left(x_{n-1} + \frac{A}{x_{n-1}}\right)$.
- b.) Estime $\sqrt{2}$ y $\sqrt[10]{1000999}$ con al menos cinco decimales exactos.

4.11 Aproxime $\sqrt[3]{0.00302}$ con al menos cinco cifras significativas

4.12 Aproxime $\sqrt[10]{0.00302}$ con al menos cinco cifras significativas

4.13 Dé una fórmula iterativa para aproximar $\sqrt[N]{a}$ con $a > 0$.

4.14 Resuelva $x^3 = 0$ usando $x_0 = -0.2$. Resuelva la misma ecuación usando bisección con el intervalo $[-0.2, 0.1]$

4.15 Resuelva $f = x^5 - 100 * x^4 + 3995 * x^3 - 79700 * x^2 + 794004 * x - 3160075$ usando $x_0 = 17$. Resuelva usando bisección con $[17, 22.2]$

4.16 Resuelva $x^3 - 2x - 5 = 0$. Esta ecuación tiene valor histórico: fue la ecuación que usó John Wallis para presentar por primera vez el método de Newton a la academia francesa de ciencias en el siglo XV.

4.17 Resuelva $e^{3(x-1)} - \ln(x-1)^2 + 1 = 0$ con al menos cinco cifras significativas.

4.18 Resuelva $e^{3x} - \ln(x^2 + 1) - 30 = 0$ con al menos cinco cifras significativas.

4.19 Considere la ecuación $x = u \ln(u)$ con $x \geq 0$.

- a.) Aplique el método de Newton a $f(u)$ y obtenga la expresión para u_{n+1} .
- b.) Resuelva la ecuación $f(u) = 0$ con $u_0 = 0.5$

4.20 $x = 2$ es un cero del polinomio $P(x) = -1536 + 6272x - 11328x^2 + 11872x^3 - 7952x^4 + 3528x^5 - 1036x^6 + 194x^7 - 21x^8 + x^9$. Aproxime esta raíz con una aproximación inicial adecuada.

4.21 $x = 1$ es un cero del polinomio $P(x) = 2 - 19x + 81x^2 - 204x^3 + 336x^4 - 378x^5 + 294x^6 - 156x^7 + 54x^8 - 11x^9 + x^{10}$. Aproxime esta raíz con una aproximación inicial adecuada.

4.22 Para las siguientes ecuaciones, haga una corrida de su programa **newtonraphson()** para las aproximaciones iniciales que se dan. ¿Es posible encontrar otra aproximación inicial para obtener convergencia eficiente?

- a.) **(Sucesión oscilante y divergente).** $\arctan(x) = 0$ con $x_0 = 1.5$

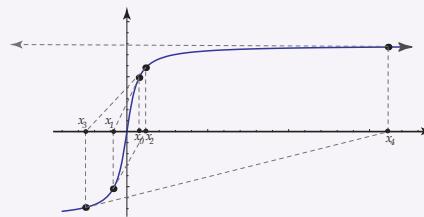


Figura 4.12: $f(x) = \arctan(x)$

- b.) **(Sucesión rápidamente convergente).** $x^2 - \cos(x) - 1 = 0$. con $x_0 = 6$

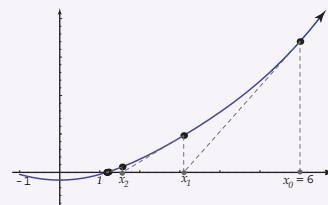


Figura 4.13: $f(x) = x^2 - \cos(x) - 1$

- c.) **Sucesión Periódica (cíclica).** $x^3 - x - 3 = 0$ con $x_0 = 0$

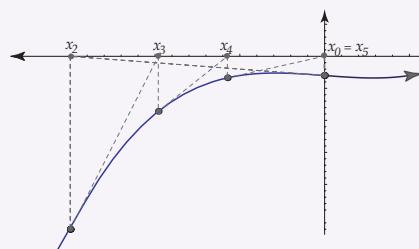


Figura 4.14: $f(x) = x^3 - x - 3$

- d.) **(Convergencia lenta).** $(x - 1)^3 = 0$ con $x_0 = -2$

4.23 Sea $f(x) = x^3 - 2\cos(x) - 3$. Según el teorema de Taylor

$$f(\alpha) = f(x) + f'(x)(\alpha - x) + \frac{1}{2}(\alpha - x)^2 f''(\xi), \text{ con } \xi \text{ entre } x \text{ y } \alpha$$

Si $\alpha = \pi/4$ y $x = 1$, calcule ξ .

- 4.24** Considere la función $f(x) = \operatorname{sgn}(x-1)\sqrt{|x-1|}$ donde $\operatorname{sgn}(u) = \begin{cases} 1 & \text{si } u > 0 \\ 0 & \text{si } u = 0 \\ -1 & \text{si } u < 0 \end{cases}$ El gráfico de la función es

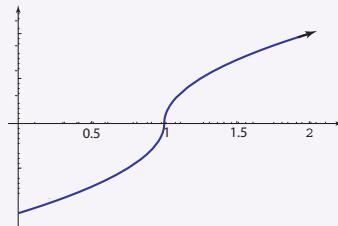


Figura 4.15: $f(x) = \operatorname{sgn}(x-1)\sqrt{|x-1|}$

Claramente $x^* = 1$ es un cero de f . Use el método de Newton e intente aproximar este cero. Comente el resultado de sus experimentos.

4.8 Un método híbrido: Newton-Bisección.

Si el método de Newton falla (en algún sentido) en una iteración, podemos usar bisección para dar un pequeño salto y regresar al método de Newton lo más pronto posible.

Supongamos que $f(a)f(b) < 0$. Sea $x_0 = a$ o $x_0 = b$. En cada iteración una nueva aproximación x' es calculada y a y b son actualizados como sigue

- a.) si $x' = x_0 - \frac{f(x_0)}{f'(x_0)}$ cae en $[a, b]$ lo aceptamos, sino usamos bisección, es decir $x' = \frac{a+b}{2}$.
- b.) Actualizar: $a' = x'$, $b' = b$ o $a' = a$, $b' = x'$, de tal manera que $f(a')f(b') \leq 0$.

Para garantizar que $x' = x_0 - \frac{f(x_0)}{f'(x_0)} \in [a, b]$, no debemos usar directamente este cálculo para evitar la división por $f'(x_0)$ (que podría causar problemas de “overflow” o división por cero). Mejor usamos un par de desigualdades equivalentes.

Observemos que $x_{n+1} \in]a_n, b_n[$ si y sólo si

$$a_n < x_n - \frac{f(x_n)}{f'(x_n)} < b_n$$

entonces

- Si $f'(x_n) > 0$

$$(a_n - x_n)f'(x_n) < -f(x_n) \text{ y } (b_n - x_n)f'(x_n) > -f(x_n)$$

- Si $f'(x) < 0$

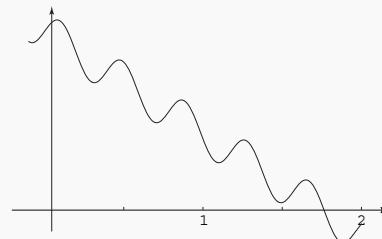
$$(a_n - x_n)f'(x_n) > -f(x_n) \text{ y } (b_n - x_n)f'(x_n) < -f(x_n)$$

En este algoritmo, se pasa a bisección si $x' = x_0 - \frac{f(x_0)}{f'(x_0)}$ sale del intervalo, pero esto no indica necesariamente que la iteración de Newton tenga algún tipo de problema en ese paso. Lo que si es importante es que se podría escoger un intervalo $[a, b]$ hasta con extremos relativos (que podrían ser mortales para el método de Newton) y el tránsito sería seguro de todas formas.

Un algoritmo similar aparece en [?] (pág. 366). En la implementación, se pasa a bisección si x' sale del intervalo o si la reducción del intervalo no es suficientemente rápida.

Ejemplo 4.17

Consideremos la función $f(x) = 0.2 \operatorname{sen}(16x) - x + 1.75$. Esta función tiene un cero en $[1, 2]$. Así que $a = 1$ y $b = 2$. Iniciamos con $x_0 = 1$. La tabla (4.3) muestra el método de Newton y Híbrido Newton-bisección aplicado a esta ecuación. El método de Newton diverge mientras que el híbrido aproxima la solución adecuadamente.



n	Newton x_i	Error Estimado	Híbrido x_i	Error Estimado	Método Usado
1	1.170357381	0.170357381	1.17035738114819	0.170357381	Newton
2	0.915271273	0.255086108	1.58517869057409	0.414821309	Bisección
3	1.310513008	0.395241735	1.79258934528705	0.207410655	Bisección
4	1.539337071	0.228824064	1.76166924922784	0.030920096	Newton
5	1.476007274	0.063329797	1.76306225245136	0.001393003	Newton
6	1.565861964	0.08985469	1.76306130340890	9.49042×10^{-7}	Newton
...
50	-2873.507697				
51	-1720.319542				

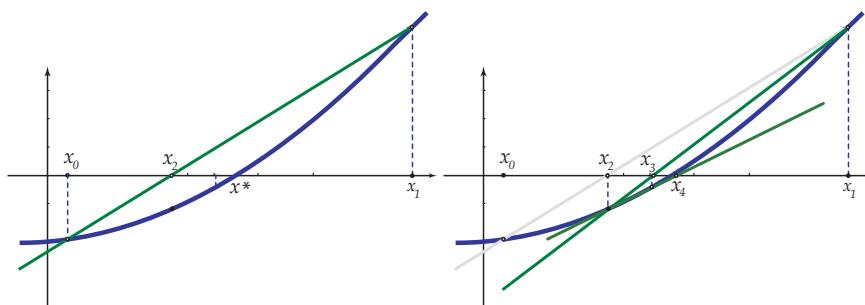
Tabla 4.3: Método de Newton e híbrido Newton-bisección aplicado a $0.2\sin(16x) - x + 1.75 = 0$ en $[1, 2]$.

Ejercicio 4.1 ■ Implementar la función `hibridoNewtonBiseccion(f,a,b, tolerancia, maxiteraciones)`

4.9 Método de la Secante

Aunque el método de la secante es anterior² al método de Newton, a veces se hace una derivación de este método basado en la iteración de Newton cambiando la derivada $f'(x_k)$ por una aproximación, lo cual puede ser muy bueno pues para algunas funciones, como las definidas por integrales o una serie, en los casos en los que la derivada no es fácil de obtener. Aquí vamos a proceder igual que antes, con una idea geométrica. El método de la secante tiene orden de convergencia de al menos $p = 1.61803$ pero en un sentido que haremos más preciso al final de esta sección, este método es más rápido que el método de Newton.

Iniciando con *dos aproximaciones iniciales* x_0 y x_1 , en el paso $k+1$, x_{k+1} se calcula, usando x_k y x_{k-1} , como la intersección con el eje X de la recta (secante) que pasa por los puntos $(x_{k-1}, f(x_{k-1}))$ y $(x_k, f(x_k))$



²El método de la secante ya era usando para calcular valores de la función seno en algunos textos Indios del siglo XV.

Figura 4.16: Método de la secante

Entonces, si $f(x_k) - f(x_{k-1}) \neq 0$,

$$x_{k+1} = \frac{x_{k-1}f(x_k) - x_kf(x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Sin embargo, para cuidarnos del fenómeno de cancelación (cuando $x_k \approx x_{k-1}$ y $f(x_k)f(x_{k-1}) > 0$), rescribimos la fórmula como

$$x_{k+1} = x_k - f(x_k) \cdot \frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}, \quad k \geq 1,$$

Aunque esta última versión no es totalmente segura, es al menos mejor que la anterior.

- Usualmente escogemos x_0 y x_1 de tal manera que el cero que queremos aproximar esté entre estos números. Si la función f es dos veces diferenciable en un entorno de un cero simple x^* se garantiza que si x_0 y x_1 se escogen “suficientemente cercanos” a x^* , entonces el método converge a x^* .

En general, si x_0 y x_1 no están suficientemente cercanos a un cero, entonces puede pasar que haya intervalos $[x_n, x_{n-1}]$ (o $[x_{n-1}, x_n]$) sin un cero en ellos. Aún así el método puede ser que converja aunque no se garantiza que sea al cero buscado.

- **Criterio de parada.** Cuando el método de la secante converge, $|x_k - x_{k-1}|$ eventualmente se vuelve pequeño. Pero en general, $|x_k - x_{k-1}|$ se mantiene bastante más grande que $|x_k - x^*|$. Si $|f'(x)| \geq m > 0$ en un intervalo I que contenga a x_k y a x^* , la estimación del error se podría hacer con $\frac{|f(x_k)|}{m}$. En todo caso, como criterio de parada podemos usar

$$|x_k - x_{k+1}| \leq \delta (|x_{k+1}| + 1) \text{ y } |f(x_{k+1})| < \epsilon$$

junto con un número máximo de iteraciones.

Ejemplo 4.18

Consideremos la ecuación $x^3 - 0.2x^2 - 0.2x - 1.2 = 0$. Esta ecuación tiene una solución en el intervalo $[1, 1.5]$. En realidad la solución es $x^* = 1.2$. Vamos a aplicar el método de la secante con $x_0 = 1$ y $x_1 = 1.5$.

$$x_0 = 1, x_1 = 1.5$$

Calcular x_2 .

$$x_2 = x_1 - f(x_1) \cdot \frac{(x_1 - x_0)}{f(x_1) - f(x_0)} = 1.1481481481481481$$

Calcular x_3 .

$$x_3 = x_2 - f(x_2) \cdot \frac{(x_2 - x_1)}{f(x_2) - f(x_1)} = 1.1875573334135374$$

Calcular x_4 .

$$x_4 = x_3 - f(x_3) \cdot \frac{(x_3 - x_2)}{f(x_3) - f(x_2)} = 1.2006283753725182$$

Calcular x_5 .

$$x_5 = x_4 - f(x_4) \cdot \frac{(x_4 - x_3)}{f(x_4) - f(x_3)} = 1.1999926413206037$$

Para estimar el error vamos a usar $|x_k - x_{k-1}|$. En la tabla (4.16) hacemos una comparación entre el error correcto para la solución $x^* = 1.2$ y la estimación del error.

k	x_k	$ x_k - x^* $	$ x_k - x_{k-1} $
1	1.5	0.3	0.5
2	1.14815	0.0518519	0.351852
3	1.18756	0.0124427	0.0394092
4	1.20063	0.000628375	0.013071
5	1.19999	7.35868×10^{-6}	0.000635734
6	1.2000000000000030		4.31745×10^{-9}

Ejemplo 4.19

Consideremos $P(x) = x^5 - 100x^4 + 3995x^3 - 79700x^2 + 794004x - 3160075$. En la figura (4.17) se muestra la gráfica de P con las primeras dos secantes usando $x_0 = 22.2$ y $x_1 = 17$.

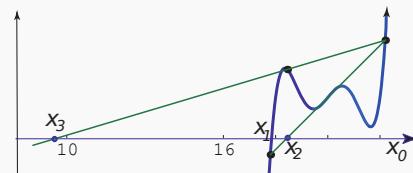


Figura 4.17

P tiene un cero, $x^* = 17.846365121\dots$, en el intervalo $[17, 22.2]$. Vamos a aproximar este cero usando $x_0 = 17$ y $x_1 = 22.2$ y también con $x_0 = 22.2$ y $x_1 = 17$.

$x_0 = 22.2$	$x_1 = 17$		$x_0 = 22.2$	$x_1 = 17$		
k	x_{k+1}	$ x_k - x_{k+1} $	$f(x_i)$	x_{k+1}	$ x_k - x_{k+1} $	$f(x_i)$
1	21.70509296	4.705	1.44	21.70509296	0.494	1.446
2	21.64664772	0.058	1.36	21.63787473	0.067	1.369
3	20.61844015	1.028	6.38	20.44319288	1.194	6.354
...						
23	17.84697705	0.013	0.02	21.2200121	0.429	3.503
24	17.84635118	0.000	-0.00	21.92553159	0.705	3.475
25	17.84636513	1.39×10^{-5}	5.79283×10^{-7}	111.120	89.194	627
26	17.84636512	1.37×10^{-8}	-1.86265×10^{-9}	21.925	89.194	3.475
...				
77				20.57831656	5110.34	6.410
78				20.57831656	1.06×10^{-14}	6.410

La elección $x_0 = 22.2$ y $x_1 = 17$ muestra ser adecuada. Nos lleva a la aproximación correcta $x_{26} = 17.84636512$. En la otra elección hay un fenómeno de cuidado: La iteración 78 podría inducirnos a error pues nos presenta a $x_{78} = 20.57831656$ como un cero aproximado con error estimado 1.06×10^{-14} pero $f(x_{78}) = 6.410!$.

Notemos que hay varios intervalos (con extremos x_k, x_{k+1}) que no contienen un cero.

Algoritmo

En este algoritmo, como criterio de parada usamos $|x_k - x_{k-1}| \leq \delta (|x_k| + 1)$ y $|f(x_k)| < \epsilon$ junto con un número máximo de iteraciones.

Algoritmo 4.2: Método de la secante.

Datos: Una función continua f , las aproximaciones iniciales x_0 y x_1 , **tol** y **maxIteraciones**

Salida: Si la iteración converge a un cero, una aproximación x_k del cero.

```

1  $j = 0;$ 
2 while  $|x_k - x_{k-1}| > \text{tol}$  y  $j < \text{maxIteraciones}$  do
3    $x_2 = x_1 - f(x_1) \cdot \frac{(x_1 - x_0)}{f(x_1) - f(x_0)};$ 
4    $x_0 = x_1;$ 
5    $x_1 = x_0;$ 
6    $j = j + 1;$ 
7 return  $x_2;$ 
```

■ **Implementación en R.** Vamos a implementar una función **secante(f, x0, x1, tolerancia, maxiteraciones)**. La función f se recibe por nombre.

■ Código R 4.7: Método de la secante

```

secante = function(f, x0, x1, tol, maxiter = 100){
  f0 = f(x0)
  f1 = f(x1)
  k = 0
  while (abs(x1 - x0) > tol && k <= maxiter ) {
    k = k+1
    pendiente = (f1 - f0)/(x1 - x0)
    if (pendiente == 0) return( cero = NA, f.cero = NA, iter = k, ErrorEst = NA)
    x2 = x1 - f1/pendiente
    f2 = f(x2)
    x0 = x1; f0 = f1
    x1 = x2; f1 = f2
    # Imprimir iteraciones
    cat(x1, x2, abs(x1-x0), "\n")
  }
  if (k > maxiter) {
    warning("No se alcanzó el número de iteraciones")
  }
  #return(list(cero=x2, f.cero=f2, iter=k, ErrorEst =abs(x2-x1)))
}

##--- Pruebas
f = function(x) x-cos(x)
plot(f,0, 1)
secante(f, 0, 2, 1e-15, 10)

#0.585454927933219 0.585454927933219 1.41454507206678
#0.717134868255196 0.717134868255196 0.131679940321977
#0.739900765490124 0.739900765490124 0.0227658972349276
#0.739081136054205 0.739081136054205 0.000819629435918068
#0.739085132495594 0.739085132495594 3.99644138893152e-06
#0.739085133215161 0.739085133215161 7.19566961571161e-10
#0.739085133215161 0.739085133215161 7.7715611723761e-16

```

4.10 El Método de la Falsa Posición

Este método es todavía más viejo que el método de Newton, de hecho aparece en textos Indios del siglo V ([?]). Su orden de convergencia no vas allá de ser lineal. En esta sección vamos a dar una breve descripción del método.

La idea de este método es calcular la recta secante que une los puntos extremos $(a_1, f(a_1))$ y $(b_1, f(b_1))$. Luego se determina el punto m en que esta recta corta el eje x y este valor entra a jugar el papel que en el método de bisección jugaba el punto medio.

La recta secante tiene que une los puntos $(a_1, f(a_1))$ y $(b_1, f(b_1))$ tiene ecuación

$$y = f(a_1) + \frac{f(b_1) - f(a_1)}{a_1 - b_1} (x - a_1)$$

Al resolver $y = 0$ se despeja el valor de x , obteniendo:

$$m = a_1 - \frac{f(a_1)(b_1 - a_1)}{f(b_1) - f(a_1)} \implies m = \frac{a_1 f(b_1) - b_1 f(a_1)}{f(b_1) - f(a_1)}$$

En este método no se conoce a priori el número de iteraciones requeridas para alcanzar la precisión deseada así que se usa un número máximo de iteraciones.

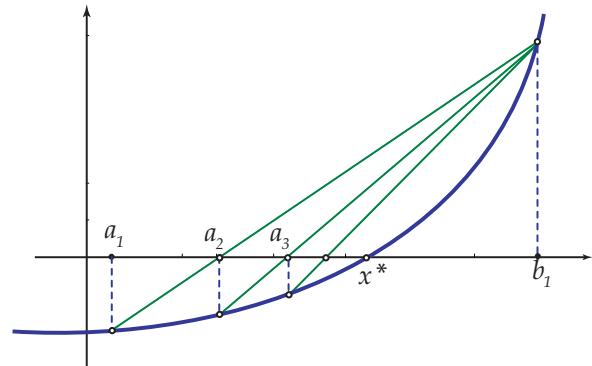


Figura 4.18: Método de la falsa posición

Algoritmo.

Algoritmo 4.3: Método de falsa posición

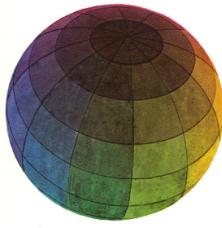
Datos: $f \in C[a, b]$ con $f(a)f(b) < 0$, δ , maxItr

Salida: Una aproximación x_n de un cero de f .

```

1  k = 0 ;
2  a_n = a, b_n = b. ;
3  repeat
4      x_n =  $\frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$  ;
5      e_1 = x_n - a_n; e_2 = b_n - x_n;
6      if f(x_n)f(a_n) > 0 then
7          a_n = x_n
8      else
9          b_n = x_n
10     k = k + 1;
11 until (máx(e_1, e_2) ≤ δ ∨ k ≥ maxItr);
12 return x_n

```



Revisado: Marzo, 2016

Versión actualizada de este libro:

<https://tecdigital.tec.ac.cr/revistamatematica/Libros/>



5 — Sistema de ecuaciones Lineales

5.1 Eliminación Gaussiana

La solución de sistemas de ecuaciones lineales es uno de los problemas básicos en métodos numéricos porque es un problema que aparece frecuentemente en la práctica, muchas veces como parte de un problema más grande.

Consideremos el sistema lineal con m ecuaciones y n indeterminadas x_1, x_2, \dots, x_n ,

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = a_{10} \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = a_{20} \\ \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = a_{m0} \end{array} \right.$$

En forma matricial se escribe: $AX = B$ donde

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix} \quad \text{y} \quad B = \begin{pmatrix} a_{10} \\ a_{20} \\ \dots \\ a_{m0} \end{pmatrix}$$

- Si $m > n$ no hay, en general solución, pero podemos aproximar una “mejor solución” minimizando una función de error (usando “mínimos cuadrados”).
- Si $m < n$ hay, en general, infinitas soluciones.

- Si $m = n$ y $\text{Det} A \neq 0$, entonces tenemos solución única. Si $B = \mathbf{0}$ solo tendríamos la solución trivial $X = \mathbf{0}$.

En este caso $n \times n$, el método de eliminación gaussiana con pivoteo parcial requiere $O\left(\frac{1}{3}n^3\right)$ operaciones.

■ **Eliminación Gaussiana.** Consideremos la matriz *ampliada*

$$A_b = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & a_{10} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{20} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & a_{m0} \end{array} \right)$$

Si F_i denota la fila i de A_b , la operaciones elementales $\alpha\bar{F}_i$, $\beta F_i + \alpha\bar{F}_j$ el intercambio de filas F_i, F_j aplicadas sobre A_b no modifican la solución del sistema $AX = b$, es decir, si A' y B' son las matrices que se obtienen después de aplicar operaciones elementales, el sistema $A'X = b'$ tiene la misma solución (si hubiera) que el sistema $AX = b$.

El algoritmo de eliminación gaussiana requiere que, cuando se está eliminando la columna j por debajo de la diagonal, el pivote a_{jj} no sea nulo, en caso contrario se hace un intercambio de fila (si hay solución única; siempre es posible hacer el cambio de fila).

Supongamos que un sistema $n \times n$ tiene solución única, entonces es posible encontrar una fila con entrada $a_{i1} \neq 0$ (en la primera columna). Supongamos que $a_{11} \neq 0$ y que la matriz ampliada del sistema $AX = B$ es

$$A_b = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array} \right)$$

El primer paso, aplicamos las operaciones $\bar{F}_i - \frac{a_{i,1}}{a_{11}} F_1$ con $i = 2, \dots, n$ y obtenemos la nueva matriz

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} & b_n^{(2)} \end{array} \right)$$

Ahora procedemos inductivamente, aplicando operaciones a la submatriz en azul. El pivote ahora es $a_{22}^{(2)} \neq 0$ (sino, intercambiamos filas).

Aplicamos las operaciones $\overline{F_i} - \frac{a_{i,2}^{(2)}}{a_{22}^{(2)}} F_2$ con $i = 3, \dots, n$ y obtenemos la nueva matriz

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & b_1 \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \dots & a_{2n}^{(2)} & b_2^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \dots & a_{3,n}^{(3)} & b_3^{(3)} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \dots & a_{nn}^{(3)} & b_n^{(3)} \end{array} \right)$$

La última operación sería (si el pivote no es nulo), $\overline{F_i} - \frac{a_{i,n-1}^{(2)}}{a_{n-1,n-1}^{(2)}} F_{n-1}$ con $i = n$ y obtenemos la matriz triangular

$$\left(\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & b_1 \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \dots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{array} \right)$$

La solución $X = (x_1, x_2, \dots, x_n)^T$ la obtenemos haciendo “sustitución hacia atrás”,

$$\begin{aligned} x_n &= \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \\ x_i &= \frac{b_i^{(i-1)} - \sum_{j=i+1}^n a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}}, \quad \text{para } i = n-1, n-2, \dots, 1. \end{aligned}$$

Ejemplo 5.1

Usar eliminación gaussiana para resolver el sistema,

$$\left\{ \begin{array}{l} 4x_1 + 2x_2 = 2 \\ 2x_1 + 3x_2 + x_3 = -1 \\ x_2 + \frac{5}{2}x_3 = 3 \end{array} \right.$$

Solución: Las primeras operaciones son $\overline{F_i} - \frac{a_{i,1}}{a_{11}} F_1$ con $i=2,3$ y la tercera es $\overline{F_3} - \frac{a_{2,1}^{(2)}}{a_{22}^{(2)}} F_2$

$$\left(\begin{array}{ccc|c} 4 & 2 & 0 & 2 \\ 2 & 3 & 1 & -1 \\ 0 & 1 & 5/2 & 3 \end{array} \right) \xrightarrow{\begin{array}{l} \overline{F_2} - \frac{1}{2} F_1 \\ \overline{F_3} - 0 F_1 \end{array}} \left(\begin{array}{ccc|c} 4 & 2 & 0 & 2 \\ 0 & 2 & 1 & -2 \\ 0 & 1 & 5/2 & 3 \end{array} \right) \xrightarrow{F_3 - \frac{1}{2} \overline{F}_2} \left(\begin{array}{ccc|c} 4 & 2 & 0 & 2 \\ 0 & 2 & 1 & -2 \\ 0 & 0 & 2 & 4 \end{array} \right)$$

Aplicando sustitución hacia atrás, la solución que encontramos es

$$\begin{aligned} x_1 &= 1.5 \\ x_2 &= -2 \\ x_3 &= 2 \end{aligned}$$

Ejemplo 5.2

Use el eliminación gaussiana con *pivoteo parcial* para resolver el sistema $AX = B$ donde

$$A = \begin{pmatrix} 0 & -1 & 4 & -1 \\ -1 & 4 & -1 & 0 \\ -1 & 0 & -1 & 4 \\ 4 & -1 & 0 & 0 \end{pmatrix} \text{ y } B = (-1, 2, 4, 10)^T$$

Solución:

F_1, F_4

$$\begin{pmatrix} 4 & -1 & 0 & 0 & 10 \\ -1 & 4 & -1 & 0 & 2 \\ -1 & 0 & -1 & 4 & 4 \\ 0 & -1 & 4 & -1 & -1 \end{pmatrix}$$

$$F_2 = F_2 - \frac{-1}{4}F_1 \text{ y } F_3 = F_3 - \frac{-1}{4}F_1$$

$$\begin{pmatrix} 4 & -1 & 0 & 0 & 10 \\ 0 & \frac{15}{4} & -1 & 0 & \frac{9}{2} \\ 0 & -\frac{1}{4} & -1 & 4 & \frac{13}{2} \\ 0 & -1 & 4 & -1 & -1 \end{pmatrix}$$

$$F_4 = F_4 - \frac{-16/15}{56/15}F_3$$

$$\begin{pmatrix} 4 & -1 & 0 & 0 & 10 \\ 0 & \frac{15}{4} & -1 & 0 & \frac{9}{2} \\ 0 & 0 & \frac{56}{15} & -1 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{26}{7} & \frac{48}{7} \end{pmatrix}$$

$$F_3 = F_3 - \frac{-1/4}{15/4}F_2 \text{ y } F_4 = F_4 - \frac{-1}{15/4}F_2$$

$$\begin{pmatrix} 4 & -1 & 0 & 0 & 10 \\ 0 & \frac{15}{4} & -1 & 0 & \frac{9}{2} \\ 0 & 0 & -\frac{16}{15} & 4 & \frac{34}{5} \\ 0 & 0 & \frac{56}{15} & -1 & \frac{1}{5} \end{pmatrix}$$

 F_3, F_4

$$\begin{pmatrix} 4 & -1 & 0 & 0 & 10 \\ 0 & \frac{15}{4} & -1 & 0 & \frac{9}{2} \\ 0 & 0 & \frac{56}{15} & -1 & \frac{1}{5} \\ 0 & 0 & -\frac{16}{15} & 4 & \frac{34}{5} \end{pmatrix}$$

Sistema

$$\left\{ \begin{array}{l} 4x_1 - x_2 = 10 \\ \frac{15}{4}x_2 - x_3 = \frac{9}{2} \\ \frac{56}{15}x_2 - x_3 = \frac{1}{5} \\ \frac{26}{7}x_4 = \frac{48}{7} \end{array} \right.$$

$$\text{Entonces } (x_1, x_2, x_3, x_4) = \left(\frac{295}{104}, \frac{35}{26}, \frac{57}{104}, \frac{24}{13} \right)$$

Ejercicio 5.1 Determine un polinomio $p(x) = a_0 + a_1x + a_2x^2$ tal que

$$\int_0^1 p(x) dx = 1$$

$$\int_0^1 x p(x) dx = \frac{7}{12}$$

$$\int_0^1 x^2 p(x) dx = \frac{13}{30}$$

■ **Implementación en R.** Por ahora vamos a implementar una versión directa del algoritmo.

■ **Código R 5.1: Eliminación gaussiana sin pivoteo parcial**

```
gauss = function(A, b){ # Se supone det(A) != 0
  n = nrow(A) # = ncol(A) para que sea cuadrada
  # matriz ampliada
  Ab = cbind(A,b)
  # Eliminación
  for (k in 1:(n-1)){
    # desde columna k=1 hasta k=n-1
    if(Ab[k,k]==0){ # intercambio de fila
      fila = which(Ab[, k]!=0)[1]
      Ab[c(k, fila), ] = Ab[c(fila, k), ]
    }
    # Eliminación columna k
    for (i in (k+1):n){# debajo de la diagonal
      #  $F_i = F_i - a_{ik}/a_{kk} * F_k$ ,  $i=k+1,\dots,n$ 
      Ab[i, ] = Ab[i, ] - Ab[i, k]/Ab[k,k]*Ab[k, ]
    }
  }
  # Sustitución hacia atrás-----
  #  $b(i) = A[i, n+1]$ 
  x = rep(NA, times=n)
  x[n] = Ab[n, n+1]/Ab[n,n]      #  $x_n = b_n/a_{nn}$ 
  for(i in (n-1):1 ){
    x[i]= (Ab[i, n+1] -sum(Ab[i, (i+1):n]*x[(i+1):n])) /Ab[i,i]
  }
  return(x)
}

#--- Pruebas
A = matrix(c( 0,  2,  3,  3,
             -5, -4,  1,  4,
              0,  0,  0,  3,
             -4, -7, -8,  9), nrow=4, byrow=TRUE)
b = c(1,0,0,0)
##  

gauss(A,b) # [1] -1.2580645  1.4193548 -0.6129032  0.0000000
solve(A,b) # [1] -1.2580645  1.4193548 -0.6129032  0.0000000
```

En la base de R está la función solve: `solve(a, b, tol, LINPACK = FALSE, ...)`. Por ejemplo,

■ **Código R 5.2: Eliminación gaussiana con solve()**

```
A = matrix(c(4, 2, 0,
            2, 3, 1,
            0, 1, 5/2), nrow=3, byrow=TRUE)
b = c(2,-1,3)

solve(A,b) # base de R

#[1] 1.5 -2.0 2.0
```

■ **Estrategia de pivoteo parcial.** Esta estrategia consiste en hacer intercambio de filas para seleccionar el pivote más grande, *en valor absoluto*. Hay matrices donde esta estrategia falla, pero es un método muy usado.

¿Cuál es la ganancia?. Cuando calculamos $A[i,j] = A[i,j] - m * A[k,j]$ con $m = A[i,k]/A[k,k]$ el error de redondeo introducido en $A[k,j]$ sería amplificado por m si $m > 1$, así que es mejor mantener m pequeño tomando el pivote $A[k,k]$ como el más grande, en valor absoluto, de la columna en las filas a eliminar. Adicionalmente, como $sols[i] = suma/A[i,i]$, si $A[i,i]$ es pequeño, cualquier error en el numerador puede ser incrementado de manera indeseable.

■ **Implementación en R.** Para la eliminación gaussiana con pivote parcial necesitamos buscar el pivote adecuado. La función `which.max(abs(x))` devuelve el índice de la entrada más grande, en valor absoluto, en el vector x . En nuestro caso, debemos buscar en la columna k , de la diagonal para abajo, es decir,

```
fila = which.max( abs(A[k:n,k]))+ k-1
```

Debemos sumar $k-1$ pues el vector $A[k:n,k]$ solo tiene $n-k+1$ componentes (no es la columna k completa).

■ Código R 5.3: Eliminación gaussiana con pivote parcial

```
gaussPP = function(A, b){
  if(is.matrix(A)) {
    n = nrow(A); m = ncol(A)
    if (m != n) stop("A' debe ser una matriz cuadrada.")
  }
  # matriz ampliada
  Ab = cbind(A,b)
  # Eliminación
  for (k in 1:(n-1)){  # desde columna k=1 hasta k=n-1
    # índice del pivote máximo, en valor absoluto
    # which.max( A[k:n,k] ) retorna índice del vector A[k:n,k] = (a_kk, a_(k+1)k, ..., a_nk)
    # Como a_kk tendría índice 1, hay que corregir el índice sumando k-1.
    fila = which.max( abs(A[k:n,k]) ) + k-1
    Ab[c(k, fila), ] = Ab[c(fila, k), ]
```

```

# Si pivote es cero, det A = 0!
if(A[fila,k]==0) stop("La matriz es singular")
# Eliminación columna k
for (i in (k+1):n){# debajo de la diagonal
  # F_i = F_i - a_ik/a_kk * F_k, i=k+1,...,n
  Ab[i, ] = Ab[i, ] - Ab[i, k]/Ab[k,k]*Ab[k, ]
}
}

# Sustitución hacia atrás-----
# b(i) = A[i, n+1]
x = rep(NA, times=n)
x[n] = Ab[n, n+1]/Ab[n,n]      # xn = bn/ann
for(i in (n-1):1){ # for
  x[i]= (Ab[i, n+1] - sum(Ab[i, (i+1):n]*x[(i+1):n])) /Ab[i,i]
}
return(x)
}

##### Pruebas
A = matrix(c( 0,  2,  3,  3,
             -5, -4,  1,  4,
              0,  0,  0,  3,
             -4, -7, -8,  9), nrow=4, byrow=TRUE)
b = c(1,0,0,0)
## gaussPP(A,b) # [1] -1.2580645  1.4193548 -0.6129032  0.0000000
solve(A,b)   # [1] -1.2580645  1.4193548 -0.6129032  0.0000000
#####

```

Ejercicio 5.2 Resolver el sistema que sigue, usando pivoteo parcial.

$$\begin{pmatrix} 14 & 14 & -9 & 3 & -5 \\ 14 & 52 & -15 & 2 & -32 \\ -9 & -15 & 36 & -5 & 16 \\ 3 & 2 & -5 & 47 & 49 \\ -5 & -32 & 16 & 49 & 79 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -15 \\ -100 \\ 106 \\ 329 \\ 463 \end{bmatrix}$$

Solución: (0, 1, 2, 3, 4)

5.2 Análisis de error.

En general hay tres fuentes de error. Primero los coeficientes de la matriz, pues trabajamos con una matriz “perturbada” (debido a errores en la toma de datos o por errores de redondeo) $A + \delta A$ en vez de la matriz exacta A . Error en el lado derecho del sistema porque trabajamos con $B + \delta b$ en vez de B y errores de redondeo porque calculamos con valores aproximados (no exactos).

Normas. La norma de una matriz va a ser útil para el análisis de error y para criterios de parada en el caso de métodos iterativos.

La norma de un vector \mathbf{x} es una función $\|\mathbf{x}\| \in \mathbb{R}$ que cumple

- a.) $\|\mathbf{x}\| > 0$ si $\mathbf{x} \neq 0$
- b.) $\|\alpha\mathbf{x}\| = |\alpha| \|\mathbf{x}\|$
- c.) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

Sea $\text{vec}\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$. Algunos ejemplos de normas son,

$$\begin{aligned} \text{a.) } \|\mathbf{x}\|_1 &= \sum_{i=1}^n |x_i| \\ \text{b.) } \|\mathbf{x}\|_2 &= \sqrt{\sum_{i=1}^n x_i^2} \\ \text{c.) } \|\mathbf{x}\|_\infty &= \max_{1 \leq i \leq n} |x_i| \end{aligned}$$

La norma de una matriz se define usando normas de vectores por cuestiones prácticas: Si $\|\cdot\|$ es una norma para vectores $\mathbf{x} \in \mathbb{R}^n$, la norma de la matriz A se define como

$$\|A\| = \max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$$

Esta manera de definir una norma matricial es práctica porque entonces se tiene

- a.) $\|AB\| \leq \|A\| \|B\|$
- b.) $\|A\mathbf{x}\| \leq \|A\| \|\mathbf{x}\|$

Ahora usando las normas que definimos anteriormente para vectores, podemos calcular las normas correspondientes para las matrices (ya como fórmulas dadas). Si $A = (a_{ij})_{n \times n}$, entonces

$$\text{a.) } \|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|$$

b.) $\|A\|_2 = \sqrt{|\sigma_{max}|}$ donde σ_i son los valores propios de $A^T A$.

c.) $\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$

Ejemplo 5.3

$$\text{Sea } A = \begin{pmatrix} 4 & -6 & 2 \\ 0 & 4 & 1 \\ 1 & 2 & 3 \end{pmatrix}.$$

La función `eigen()` de la base de **R** calcula los valores propios de una matriz. Para calcular $\|A\|_2$ usamos el código `eigen(t(A) %*% A)$values` lo que nos da

[1] 66.6816247 19.8065379 0.5118373

Entonces $\|A\|_2 = \sqrt{66.6816}$. También $\|A\|_\infty = 12$

Sistemas perturbados. Consideremos el sistema de ecuaciones perturbado

$$(A + \delta A)(X + \delta X) = b + \delta b$$

donde X es la solución exacta y δX sería un error introducido en la solución del sistema. Por ejemplo, sea

$$\delta A_{3 \times 3} = \mathbf{0}, \quad \delta X = \mathbf{0}, \quad b = \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix} \quad \text{y} \quad \delta b = \begin{pmatrix} 0 \\ 0.0001 \\ 0 \end{pmatrix}$$

$$\text{El sistema perturbado sería } A_{3 \times 3}X = b + \delta b = \begin{pmatrix} 0 \\ 1.0001 \\ 3 \end{pmatrix}$$

Una matriz está “mal condicionado” (“ill-conditioned”) si la solución del sistema $AX = b$ cambia mucho cuando se hacen prequeños cambios (perturbaciones) en el sistema. El método de eliminación gaussiana puede ser afectado por errores de redondeo, especialmente si la matriz es “mal condicionada”

Ejemplo 5.4

$$\text{Considere el sistema lineal} \quad \begin{cases} 1.002x_1 + x_2 = 2.002 \\ x_1 + 0.998x_2 = 1.998 \end{cases}$$

La solución exacta es $X = (1, 1)^T$. Ahora consideremos las perturbaciones $\delta A = \mathbf{0}$, $\delta X = \mathbf{0}$ y $\delta b = (0.0001, 0)^T$. El sistema queda,

$$\begin{cases} 1.002x_1 + x_2 = 2.0021 \\ x_1 + 0.998x_2 = 1.998 \end{cases}$$

y la solución es ahora $(-23.95, 26.00)^T$, es decir, una pequeña perturbación en el sistema provocó un gran cambio en la solución.

Se puede estimar el error δX usando el llamado “número de condición” de A . Este número se define como $\text{cond}(A) = \|A\| \|A^{-1}\|$.

Si $e_1 = \|\delta A\|/\|A\|$, $e_2 = \|\delta b\|/\|B\|$ y $\varepsilon = \|\delta X\|/\|X\|$, entonces

$$\varepsilon \leq \frac{\text{cond}(A)}{1 - e_1 \text{cond}(A)} (e_1 + e_2)$$

Si el número de condición es grande, no significa que ε sea grande, pero si dice que el problema está mal condicionado y esto nos hace sospechar que la formulación del problema no es correcta.

`rcond(A) ≥ 1` y el recíproco del número de condición es $\text{rcond}(A) = \frac{1}{\text{cond}(A)}$. En la base de **R**, se aproxima este número con la función **rcond()**.

Si A está bien condicionada, $\text{rcond}(A)$ está cerca de 1. Si A está mal condicionado, $\text{rcond}(A)$ está cerca de 0.

En el ejemplo 5.4 se obtiene $\text{rcond}(A) = 9.98003 \times 10^{-7}$, que es muy cercano a cero.

Ejemplo 5.5

Considere el sistema lineal $A_b = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 23/12 \\ 43/30 \end{bmatrix}$

La solución exacta de este sistema es $X = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

En este caso, $\text{rcond}(A) = 0.03103147$ que es pequeño. Si agregamos la perturbación $\delta b = (0, 0.001, 0)$ el resultado es $(0.964, 2.192, 2.820)^T$

5.3 Descomposición LU.

En el ejemplo 5.1 aplicamos eliminación gaussiana a la matriz a $A = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 5/2 \end{pmatrix}$.

Aplicamos la operación $\bar{F}_2 - \frac{1}{2}F_1$ para eliminar en la columna 1, fila 2 y no necesitamos operación para eliminar en la fila 3.

$$\begin{matrix} 1/2 \\ 0 \end{matrix} \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 5/2 \end{pmatrix}$$

Luego aplicamos la operación $\bar{F}_3 - \frac{1}{2}F_2$ para eliminar en la columna 2, fila 3,

$$\begin{matrix} 1/2 \\ 0 \end{matrix} \begin{matrix} 1/2 \\ 1/2 \end{matrix} \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

Observe ahora que

$$\begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 0 & 1/2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 5/2 \end{pmatrix}$$

Si se puede factorizar $A_{n \times n}$ como $A = LU$ donde L es triangular superior (con 1's en la diagonal) y U es triangular inferior, entonces el sistema $AX = b$ se puede resolver usando L y U de la siguiente manera,

a.) Resolvemos $LY = b$ y obtenemos la solución Y^*

b.) Resolvemos $UX = Y^*$ y obtenemos la solución X^* del sistema

X^* es la solución del sistema original pues como $UX^* = Y^*$ entonces $LY^* = LUX^* = AX^* = b$.

La ganancia es que si tenemos la descomposición LU de A , entonces el sistema $AX = b$ lo podemos resolver para distintas matrices b sin tener que estar aplicando el método de eliminación gaussiana cada vez desde el principio. Observemos que los sistemas $LY = b$ y $UX = Y^*$ son sistemas sencillos pues L y U son matrices triangulares.

Ejemplo 5.6

Resolver, usando la descomposición LU , es sistema $A = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 5/2 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 4 \end{bmatrix}$.

Solución: Una descomposición LU de A es

$$\begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 0 & 1/2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

Entonces $L = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 0 & 1/2 & 1 \end{pmatrix}$ y $U = \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$

Luego resolvemos $LY = b$, es decir, $\begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 0 & 1/2 & 1 \end{pmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 4 \end{bmatrix}$ y nos da $Y^* = (2, 3, 2.5)^T$.

Ahora $UX = Y^*$, es decir, $\begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 2 & 2 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 2.5 \end{bmatrix}$ y obtenemos $X^* = (0.0625, 0.8750, 1.2500)^T$.

■ **Algoritmo.** Podemos usar la idea del ejemplo con el que iniciamos esta sección. Sea $m_{i1} = \frac{a_{i1}}{a_{11}}$ con $i > 1$, entonces la primera columna se elimina con las operaciones $\bar{F}_i - m_{i1} F_1$ con $i = 2, \dots, n$. Pero esto es equivalente a multiplicar las matrices E_1 y A , donde

$$E_1 = I - \begin{pmatrix} 0 & 0 & \dots & 0 \\ m_{21} & 0 & \dots & 0 \\ m_{31} & 0 & \dots & 0 \\ \vdots & 0 & \ddots & 0 \\ m_{n1} & 0 & \dots & 0 \end{pmatrix}$$

es decir,

$$E_1 A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix}$$

Ahora si $m_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}$ con $i > 2$, entonces

$$E_2 = I - \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & m_{22} & 0 & \dots & 0 \\ 0 & m_{32} & 0 & \dots & 0 \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & m_{n2} & 0 & \dots & 0 \end{pmatrix}$$

y se tiene

$$E_2 E_1 A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \dots & a_{3,n}^{(3)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \dots & a_{nn}^{(3)} \end{pmatrix}$$

Algoritmo. El algoritmo lo podemos establecer con ayuda de las matrices $R_k = \left(r_{ij}^{(k)} \right)$ donde las entradas de R_k se definen como,

$$r_{ij}^{(k)} = \begin{cases} \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} & \text{si } j = k \text{ e } i > k \\ 0 & \text{en otro caso.} \end{cases}$$

Es decir, las entradas de R_k son nulas excepto talvez las entradas $r_{ik}^{(k)} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$ con $i = k+1, \dots, n$

Con esta notación,

$$E_k = I - R_k$$

$$U = E_n E_{n-1} \cdots E_2 E_1 A \quad \text{y} \quad L = (E_n E_{n-1} \cdots E_2 E_1)^{-1} = I + R_1 + R_2 + \dots + R_{n-1}$$

Para resolver sistemas lineales con la descomposición LU usamos una sola matriz $\begin{pmatrix} & U \\ L & \end{pmatrix}$.

Por ejemplo, para resolver el sistema $AX = b$ con $A = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 5/2 \end{pmatrix}$ usamos directamente la matriz $A = \begin{pmatrix} 4 & 2 & 0 \\ 1/2 & 2 & 1 \\ 0 & 1/2 & 2 \end{pmatrix}$

Una implementación en **R** se muestra en el código que sigue,

■ Código R 5.4: Descomposición LU, sin pivoteo.

```
luA = function(A){
  n = nrow(A)
  LU = A
  for(j in 1:(n-1)){
    # columna j
    for(i in (j+1): n){ # filas i > j
      LU[i,j]=LU[i,j]/LU[j,j]           # Construye de L
      for(k in (j+1):n){ # Eliminación filas i > j. Pivote LU[j,j]
        LU[i,k] = LU[i,k] - LU[i,j]*LU[j,k] # Construye U
      }
    }
  }
  return(LU)
}
## ---
A = matrix(c(4, 2, 0,
            2, 3, 1,
            0, 1, 5/2), nrow=3, byrow=TRUE)

luA(A)
#      [,1] [,2] [,3]
# [1,] 4.0  2.0  0
# [2,] 0.5  2.0  1
# [3,] 0.0  0.5  2
```

Ahora podemos implementar la sustitución hacia atrás usando la parte apropiada de la matriz “**LU**”

■ Código R 5.5: Solución usando $A=LU$, sin pivoteo – versión no vectorizada.

```
# Usa la descomposición LU de A para resolver sistemas AX=b
solvelu = function(LU,b){
```

```

n = nrow(LU)
sol = rep(NA, times=n)
# Sustitución hacia atrás -----
sol[1] = b[1]
for(i in 2:n){
  s=0
  for(j in 1: (i-1)){
    s = s + LU[i,j]*sol[j]
  }
  sol[i] = b[i] - s
}
# Resolver UX=Y
sol[n] = sol[n]/LU[n,n]
for(i in (n-1): 1{
  s = 0
  for(j in (i+1): n){
    s = s + LU[i,j]*sol[j]
  }
  sol[i] = (sol[i] - s)/LU[i,i]
}
return(sol)
}

## pruebas-----
A = matrix(c(4, 2, 0,
            2, 3, 1,
            0, 1, 5/2), nrow=3, byrow=TRUE)
b = c(2,-1,3)
# Descomposición LU de A
LU = luA(A)
# Resolver sistemas AX = b usando la descomposición LU
solvelu(LU,b) # [1] 1.5 -2.0 2.0

```

■ **Existencia de la factorización LU.** La factorización de la matriz $A_{n \times n}$, no-singular, existe (y es única) si y sólo si las submatrices principales A_i , con $i = 1, 2, \dots, n-1$ son no-singulares (A_i es A con una poda: Solo se dejan las primeras i filas y las primeras i columnas).

No podemos aplicar el algoritmo de descomposición LU a la matriz

$$A = \begin{pmatrix} 1 & 1 & 3 \\ 2 & 2 & 2 \\ 3 & 6 & 4 \end{pmatrix}$$

pues habría divisiones por cero. Por ejemplo,

```
A = matrix(c(1, 1, 3,
            2, 2, 2,
            3, 6, 4), nrow=3, byrow=TRUE)
luA(A)

#      [,1] [,2] [,3]
# [1,]    1    1    3
# [2,]    2    0   -4
# [3,]    3  Inf  Inf  # divisiones por cero
```

Aquí el problema es que la matriz $A_2 = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$ es singular ($\det(A_2) = 0$).

Las matrices A_i son invertibles por ejemplo cuando A es simétrica y “definida positiva”, es decir, A es simétrica y $x^T A x > 0$ para todo $x \in \mathbb{R}^n$. También las matrices A_i son invertibles cuando A es *diagonalmente dominante* por filas o por columnas, es decir,

$$|a_{ii}| \geq \sum_{j=1, j \neq i}^n |a_{ij}|, \quad i = 1, 2, \dots, n \quad (\text{dominante por filas})$$

$$|a_{ii}| \geq \sum_{j=1, j \neq i}^n |a_{ji}|, \quad i = 1, 2, \dots, n \quad (\text{dominante por columnas})$$

■ **Pivoteo parcial en la descomposición LU.** Si A es una matriz cuadrada no singular, podemos obtener una descomposición aplicando pivoteo parcial en la descomposición LU , solo hay que llevar un registro del intercambio de filas con tal de aplicar de manera apropiada la sustitución hacia atrás.

Por ejemplo, apliquemos la descomposición LU con pivoteo parcial, a la matriz

$$A = \begin{pmatrix} 4 & 0 & 1 & 1 \\ 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{pmatrix}$$

Iniciamos con pivote $a_{11} = 4$.

#L

```
[,1] [,2] [,3] [,4]
[1,] 0.00  0   0   0
[2,] 0.75  0   0   0
[3,] 0.00  0   0   0
[4,] 0.75  0   0   0
```

#U

```
[,1] [,2] [,3] [,4]
[1,] 4    0 1.00 1.00
[2,] 0    1 2.25 0.25
[3,] 0    1 2.00 0.00
[4,] 0    2 3.25 0.25
```

Intercambio de filas $F2, F4$ (en ambas, L y U)

#L

```
[,1] [,2] [,3] [,4]
[1,] 0.00  0   0   0
[2,] 0.75  0   0   0
[3,] 0.00  0   0   0
[4,] 0.75  0   0   0
```

#U

```
[,1] [,2] [,3] [,4]
[1,] 4    0 1.00 1.00
[2,] 0    2 3.25 0.25
[3,] 0    1 2.00 0.00
[4,] 0    1 2.25 0.25
```

Pivote $a_{22}^{(2)} = 2$.

#L

```
[,1] [,2] [,3] [,4]
[1,] 0    0.0  0   0
[2,] 0.75 0.0  0   0
[3,] 0    0.5  0   0
[4,] 0.75 0.5  0   0
```

#U

```
[,1] [,2] [,3] [,4]
[1,] 4    0 1.000 1.000
[2,] 0    2 3.250 0.250
[3,] 0    0 0.375 -0.125
[4,] 0    0 0.625 0.125
```

Intercambio de filas $F3, F4$ (en ambas, L y U).

#L

```
[,1] [,2] [,3] [,4]
[1,] 0    0.0  0   0
[2,] 0.75 0.0  0   0
[3,] 0.75 0.5  0   0
[4,] 0    0.5  0   0
```

#U

```
[,1] [,2] [,3] [,4]
[1,] 4    0 1.000 1.000
[2,] 0    2 3.250 0.250
[3,] 0    0 0.625 0.125
[4,] 0    0 0.375 -0.125
```

Pivote $a_{33}^{(3)} = 0.625$

#L	[,1]	[,2]	[,3]	[,4]
[1,]	0	0.0	0	0
[2,]	0.75	0.0	0	0
[3,]	0.75	0.5	0	0
[4,]	0	0.5	0.6	0

#U	[,1]	[,2]	[,3]	[,4]
[1,]	4	0	1.000	1.000
[2,]	0	2	3.250	0.250
[3,]	0	0	0.625	0.125
[4,]	0	0	0.000	-0.200

Entonces, la descomposición LU con pivoteo parcial es

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.75 & 1 & 0 & 0 \\ 0.75 & 0.5 & 1 & 0 \\ 0 & 0.5 & 0.6 & 1 \end{pmatrix} \quad y \quad U = \begin{pmatrix} 4 & 0 & 1 & 1 \\ 0 & 2 & 3.250 & 0.250 \\ 0 & 0 & 0.625 & 0.125 \\ 0 & 0 & 0 & -0.200 \end{pmatrix}$$

Ahora observamos que $LU = \begin{pmatrix} 4 & 0 & 1 & 1 \\ 3 & 2 & 4 & 1 \\ 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \neq A$. Pero si aplicamos a I_4 los intercambios de fila que aplicamos (matriz de permutación), entonces $LU = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}^T A$

Ejemplo 5.7

Considere la matriz $A = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 1 & 5 & 1 \\ 1 & 0 & 1 & 4 \end{pmatrix}$

- (a.) Calcule una descomposición LU de A , con *pivote parcial*.

Solución:

$L:$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 \end{pmatrix}$$

$$F_2 = F_2 - \frac{1}{4}F_1 \text{ y } F_4 = F_4 - \frac{1}{4}F_1$$

$$\begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & \frac{23}{4} & 1 & 0 \\ 0 & 1 & 5 & 1 \\ 0 & -\frac{1}{4} & 1 & 4 \end{pmatrix}$$

 $L:$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{4}{23} & 0 & 0 \\ \frac{1}{4} & \frac{-1}{23} & 0 & 0 \end{pmatrix}$$

$$F_3 = F_3 - \frac{4}{23}F_2 \text{ y } F_4 = F_4 - \frac{-1}{23}F_2$$

$$\begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & \frac{23}{4} & 1 & 0 \\ 0 & 0 & \frac{111}{23} & 1 \\ 0 & 0 & \frac{24}{23} & 4 \end{pmatrix}$$

 $L:$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{4}{23} & 0 & 0 \\ \frac{1}{4} & \frac{-1}{23} & \frac{8}{37} & 0 \end{pmatrix}$$

$$F_4 = F_4 - \frac{8}{37}F_3$$

$$\begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & \frac{23}{4} & 1 & 0 \\ 0 & 0 & \frac{111}{23} & 1 \\ 0 & 0 & 0 & \frac{140}{37} \end{pmatrix}$$

Por tanto, $L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{4} & 1 & 0 & 0 \\ 0 & \frac{4}{23} & 1 & 0 \\ \frac{1}{4} & \frac{-1}{23} & \frac{8}{37} & 1 \end{pmatrix}$ y $U = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & \frac{23}{4} & 1 & 0 \\ 0 & 0 & \frac{111}{23} & 1 \\ 0 & 0 & 0 & \frac{140}{37} \end{pmatrix}$

(b.) Use la descomposición LU anterior para resolver el sistema $AX = B$ con $B = (1, 7, 16, 14)^T$.

Solución:

$$a) LY = B \implies \begin{cases} y_1 &= 1 \\ \frac{1}{4}y_1 + y_2 &= 7 \\ \frac{4}{23}y_2 + y_3 &= 16 \\ \frac{1}{4}y_1 - \frac{1}{23}y_2 + \frac{8}{37}y_3 + y_4 &= 14 \end{cases} \implies Y = \left(1, \frac{27}{4}, \frac{341}{2}, \frac{401}{37}\right)$$

$$b) UX = Y \implies \begin{cases} 4x_1 + x_2 &= 1 \\ \frac{23}{4}x_2 + x_3 &= \frac{27}{4} \\ \frac{111}{23}x_3 + x_4 &= \frac{341}{2} \\ \frac{140}{37}x_4 &= \frac{401}{37} \end{cases} \implies X = \left(\frac{9}{140}, \frac{26}{35}, \frac{347}{140}, \frac{401}{140} \right)$$

■ **Paquete Matrix.** La función **lu()** del paquete **Matrix** hace la descomposición *LU* aplicando pivoteo parcial.

```
# install.packages("Matrix")
require(Matrix)
A = matrix(c(4, 0, 1, 1,
            3, 1, 3, 1,
            0, 1, 2, 0,
            3, 2, 4, 1), nrow=4, byrow=TRUE)

mlu = lu(A)
mlu = expand(mlu)
mlu$L
#      [,1] [,2] [,3] [,4]
#[1,] 1.00   .    .    .
#[2,] 0.75  1.00   .    .
#[3,] 0.75  0.50  1.00   .
#[4,] 0.00  0.50  0.60  1.00

mlu$U
#      [,1] [,2] [,3] [,4]
#[1,] 4.000 0.000 1.000 1.000
#[2,]   .    2.000 3.250 0.250
#[3,]   .    .    0.625 0.125
#[4,]   .    .    .   -0.200

mlu$P
#4 x 4 sparse Matrix of class "pMatrix"
#[1,] | . . .
#[2,] . . | .
#[3,] . . . |
#[4,] . | . .

#PLU = A
mlu$P%*%mlu$L%*%mlu$U
4 x 4 Matrix of class "dgeMatrix"
[,1] [,2] [,3] [,4]
[1,] 4    0    1    1
```

$$\begin{bmatrix} 2, \\ 3, \\ 4, \end{bmatrix} \begin{matrix} 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{matrix}$$

Para cualquier $A_{n \times n}$ no-singular, se puede obtener una factorización LU permutando de manera adecuada las filas. Si hacemos pivoteo parcial, en general la descomposición LU que encontramos no es igual a A , pero podemos usar una matriz de permutación adecuada P y obtenemos $P^T A = LU$. Para obtener P , iniciamos con I y aplicamos a I las permutaciones de fila que aplicamos en A . Al final resolvemos $LY = P^T b$ y $UX = Y$ y esto nos da la solución del sistema usando pivoteo parcial.

Lo primero que hay que notar es que si P es una matriz de permutación, entonces $P^{-1} = P^T$, de esta manera si usamos pivoteo parcial, $PLU = A$ y

$$AX = b \implies LUX = b \implies LUX = P^T b$$

por lo que resolver $AX = b$ con descomposición LU con pivoteo parcial solo requiere *registrar los cambios de fila y aplicarlos a b*.

El procedimiento es como antes, primero obtenemos la solución Y^* del sistema $LY = P^T b$ y luego resolvemos $UX = Y^*$.

En el ejemplo anterior, podemos restablecer A multiplicando por la matriz de permutación P^T : Los cambios de fila en el pivoteo fueron: En la eliminación de la segunda columna se intercambió F_2, F_3 en la eliminación de la tercera columna se intercambió F_3, F_4 . Aplicamos a I esos mismos intercambios, obtenemos P :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{\substack{F_2, F_3 \\ F_3, F_4}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = P$$

Ahora, para resolver el sistema

$$\begin{pmatrix} 4 & 0 & 1 & 1 \\ 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

usando descomposición LU (con pivoteo parcial) de A , es equivalente a resolver el sistema

$$LU \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = P^T \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_3 \\ b_4 \\ b_2 \end{bmatrix}$$

Ejercicio 5.3 Resolver, usando descomposición LU con pivoteo parcial, los sistemas $AX = b$, $AX = b'$ y $AX = b''$ donde

$$A = \begin{pmatrix} 4 & 0 & 1 & 1 \\ 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{pmatrix}, \quad b = (5, 6, 13, 1)^T, \quad b' = (6, 7, 8, 9)^T \quad \text{y} \quad b'' = (1, 2, 5, 2)^T$$

5.4 Refinamiento iterativo.

La eliminación gaussiana puede ser afectada fuertemente por errores de redondeo, sobre todo si la matriz A es “mal condicionada”. En algunos casos (si $\text{rcon}(A)$ no es demasiado pequeño), se puede refinar una solución X_c obtenida al resolver el sistema $AX = b$ con un algoritmo llamado “refinamiento iterativo”. La idea es esta: Si X_c es solución del calculada del sistema $AX = b$ y si X^* es la solución exacta, entonces el error $e = X^* - X_c$ también es solución del sistema $Ae = R$ donde $R = b - AX_c$. Por supuesto, no conocemos e , pero lo podemos aproximar y corregir la solución X_C . El algoritmo es como sigue,

- a.) $X^{(0)}$ es la solución calculada al resolver $AX = b$ la primera vez, con la descomposición LU con pivoteo parcial.

Refinamiento iterativo.

- b.) Calcule $R^{(k)} = b - AX^{(k)}$ (Usar A !)

- c.) Resuelva $Ae = R^{(k)}$ usando la factorización LU que ya tenemos.

- d.) $X^{(k+1)} = X^{(k)} + e$

- e.) Detenerse cuando se considere que se ha alcanzado suficiente precisión.

5.5 Grandes sistemas y métodos iterativos.

Si una matriz es muy grande y *rala* (los elementos no nulos son una fracción pequeña de la totalidad de elementos de la matriz) entonces el método de eliminación gaussiana no es el mejor método para resolver el sistema porque, en general, tendríamos problemas de lentitud en el manejo de la memoria por tener que almacenar y manipular toda la matriz; además la descomposición LU deja de ser tan rala como la matriz A .

Los métodos iterativos se usan para resolver problemas con matrices muy grandes, ralas y con ciertos patrones (“bandadas”), usualmente problemas que aparecen al *discretizar* ecuaciones diferenciales en derivadas parciales.

La idea general de los métodos iterativos es, dado el sistema $AX = b$ con A no singular, descomponemos A como $A = M - N$ tal que $MY = Z$ sea “fácil” de resolver. Entonces,

$$AX = b \implies (M - N)X = b \implies MX = NX + B$$

Como $MY = z$ es “fácil” de resolver, es probable que $Y = M^{-1}z$ sea fácil de calcular, entonces

$$X = M^{-1}NX + M^{-1}B$$

Esto sugiere que si tenemos una aproximación inicial $X^{(0)}$, podemos calcular una sucesión de vectores

$$X^{(k+1)} = M^{-1}NX^{(k)} + M^{-1}B = Z^{(k)} + W^{(k)} \quad \text{donde } MZ^{(k)} = NX^{(k)} \quad \text{y } MW^{(k)} = b \quad (*)$$

hasta que la sucesión converja. Este método funciona cuando $M \approx A$.

Teorema 5.1

Sea $A_{n \times n}$ una matriz real. Sea $T = M^{-1}N$ donde $A = M - N$. Entonces la iteración (*) converge para cualquier $X^{(0)}$ si y sólo si existe una norma $\|\cdot\|$ para la que $\|T\| < 1$. Además tal norma existe si y sólo si, en valor absoluto, el más grande valor propio de T , denotado $\rho(T)$, es < 1 .

En resumen, la iteración converge si y sólo si $\rho(T) < 1$.

■ **Iteración de Jacobi.** Este el método es el más simple y el más lento. Lo que hace es descomponer A usando su diagonal D (siempre y cuando sea invertible)

$$A = D - (D - A)$$

Como $M = D$ y $N = D - A$, la iteración es

$$X^{(k+1)} = (I - D^{-1}A)X^{(k)} + D^{-1}b = X^{(k)} - D^{-1}(AX^{(k)}) + D^{-1}b$$

Iteración de Jacobi. D invertible.

$$X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)}$$

donde $Z^{(k)}$ y $W^{(k)}$ se obtienen como solución de los sistemas $DZ^{(k)} = AX^{(k)}$ y $DW^{(k)} = b$

La iteración se podría detener en $X^{(k+1)}$ si $\frac{\|X^{(k+1)} - X^{(k)}\|_\infty}{\|X^{(k+1)}\|_\infty} < tol$.

■ **Iteración de Gauss-Seidel.** Este esquema iterativo se obtiene descomponiendo A como

$$A = L_A - (L_A - A) \quad \text{donde} \quad L_A = \begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{pmatrix}$$

Como $M = L_A$ y $N = L_A - A$, la iteración es

$$X^{(k+1)} = X^{(k)} - L_A^{-1}(AX^{(k)}) + L_A^{-1}B$$

Como L_A es triangular, en vez de calcular L_A^{-1} , observamos que

$$L_A Z = AX^{(k)} \implies Z = L_A^{-1}(AX^{(k)}) \quad \text{y que} \quad L_A W = b \implies W = L_A^{-1}B$$

así que $L_A^{-1}(AX^{(k)})$ lo obtenemos como solución del sistema $L_A Z = AX^{(k)}$ y $L_A^{-1}B$ lo obtenemos como la solución del sistema $L_A W = b$. De este modo podemos escribir el esquema iterativo como

Iteración Gauss-Seidel

Sistema $AX = B$ con aproximación inicial $X^{(0)}$.

$$X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)}$$

donde cada $Z^{(k)}$ lo obtenemos como solución del sistema $L_AZ = AX^{(k)}$ y $W^{(k)}$ lo obtenemos como la solución del sistema $L_AW = B$.

Teorema 5.2 (Convergencia – Jacobi y Gauss–Seidel)

Si A es diagonalmente dominante, entonces ambos métodos, Jacobi y Gauss–Seidel convergen y, además Gauss–Seidel es más rápido. También si A es simétrica definida positiva, ambos métodos convergen.

Ejemplo 5.8

- Usando $X_0 = (0, 1, 0, 3)$, hacer dos iteraciones con el método Gauss–Seidel e indique el error estimado en cada iteración usando la norma $\| \cdot \|_\infty$

Solución:

$$L_A = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 1 & 6 & 0 & 0 \\ 0 & 1 & 5 & 0 \\ 1 & 0 & 1 & 4 \end{pmatrix} \quad \text{y} \quad X_0 = (0, 1, 0, 3).$$

a) Iteración 1.

$$L_A Z_0 = A X_0 \implies \begin{cases} 4z_1 & = 1 \\ z_1 + 6z_2 & = 6 \\ z_2 + 5z_3 & = 4 \\ z_1 + z_3 + 4z_4 & = 12 \end{cases} \implies Z_0 = \left(\frac{1}{4}, \frac{23}{24}, \frac{73}{120}, \frac{1337}{480} \right)$$

$$L_A W_0 = B \implies \begin{cases} 4w_1 & = 1 \\ w_1 + 6w_2 & = 7 \\ w_2 + 5w_3 & = 16 \\ w_1 + w_3 + 4w_4 & = 14 \end{cases} \implies W_0 = \left(\frac{1}{4}, \frac{9}{8}, \frac{119}{40}, \frac{431}{160} \right)$$

$$X_1 = X_0 - Z_0 + W_0 = \left(0, \frac{7}{6}, \frac{71}{30}, \frac{349}{120} \right)$$

$$\text{Error} \leq \frac{\|X_0 - X_1\|_\infty}{\|X_1\|_\infty} = 0.813754$$

b) **Iteración 2.**

$$L_A Z_1 = AX_1 \implies \begin{cases} 4z_1 &= \frac{7}{6} \\ z_1 + 6z_2 &= \frac{281}{30} \\ z_2 + 5z_3 &= \frac{1909}{120} \\ z_1 + 4z_4 &= 14 \end{cases} \implies Z_1 = \left(\frac{7}{24}, \frac{121}{80}, \frac{691}{240}, \frac{2599}{960} \right)$$

$$W_1 = \left(\frac{1}{4}, \frac{9}{8}, \frac{119}{40}, \frac{431}{160} \right)$$

$$X_2 = X_1 - Z_1 + W_1 = \left(-\frac{1}{24}, \frac{187}{240}, \frac{197}{80}, \frac{2779}{960} \right)$$

$$\text{Error} \leq \frac{\|X_1 - X_2\|_\infty}{\|X_2\|_\infty} = 0.133861$$

■ **Iteración SOR (successive over-relaxation).** Esta iteración es una mejora sustancial respecto a la iteración Gauss-Seidel, pero requiere la escogencia de un parámetro ω .

Sea D la diagonal de A y L_A y U_A las partes (estRICTAMENTE) triangular superior e inferior de A , es decir, L_A y U_A tienen ceros en su diagonal.

$$L_A = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ a_{21} & 0 & 0 & \dots & 0 \\ a_{31} & a_{32} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{n,(n-1)} & 0 \end{pmatrix}$$

Entonces dado ω , descomponemos A como

$$A = \left(\frac{1}{\omega} D + L_A \right) - \left(\left(\frac{1}{\omega} - 1 \right) D - U_A \right)$$

Esto nos lleva a la iteración,

$$X^{k+1} = X^k - Q_\omega^{-1} A X^k + Q_\omega^{-1} B$$

donde

$$Q_\omega = \left(\frac{1}{\omega} D + L_A \right) = \omega^{-1} (D + \omega L_A)$$

que es triangular inferior. Resumiendo,

$$X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)} \quad \text{donde resolvemos } Q_\omega Z^{(k)} = AX^{(k)} \quad \text{y} \quad Q_\omega W^{(k)} = b$$

Teorema 5.3 (Convergencia de SOR)

Si $A_{n \times n}$ es una matriz real, simétrica y semidefinida positiva, entonces para cualquier adivinanza inicial $X^{(0)} \in \mathbb{R}^n$ y para cualquier $B \in \mathbb{R}^n$ y para cualquier $\omega \in]0, 2[$, la iteración SOR converge a la solución exacta del sistema $AX = b$

Ejemplo 5.9

Considere la matriz

$$A = \begin{pmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ -1 & 0 & -1 & 4 \end{pmatrix}$$

- a.) Calcule la descomposición LU de A (con pivoteo parcial).

Solución: Aplicando nuestro algoritmo, con pivoteo parcial, obtenemos,

```
# L =
 [,1]      [,2]      [,3]      [,4]
 [1,]  1
 [2,] -0.25
 [3,]  0
 [4,] -0.25000000 -0.06666667 -0.28571429  1

# U =
 [,1]      [,2]      [,3]      [,4]
 [1,]  4.      -1       0       0
 [2,]      .    3.750000 -1.      0
 [3,]      .      .    3.733333 -1.
```

[4,]**3.714286**

- b.) Use la descomposición LU para resolver $AX = B$ con $B = (-1, 2, 4, 10)^T$.

Solución:

- Primero resolvemos $LY = B$ y obtenemos la solución $Y^* = (-1, 1.75, 4.466667, 11.142857)$
- Luego resolvemos $UX = Y^*$ y obtenemos $X = (0, 1, 2, 3)$
- Como no hubo cambios de fila, la solución es $X = (0, 1, 2, 3)$.
- Haga dos iteraciones a mano usando el método de Jacobi, con $X^{(0)} = (0, 0, 0, 0)$

Solución: La iteración es $X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)}$ donde $Z^{(k)}$ y $W^{(k)}$ son solución de los sistemas $DZ^{(k)} = AX^{(k)}$ y $DW^{(k)} = B$

- Tenemos la aproximación inicial $X^{(0)} = (0, 0, 0, 0)$.

b.) La matriz diagonal es $D = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$. Esta matriz es invertible.

- Primera iteración.

- Debemos resolver $DZ^{(0)} = A(X^{(0)})^T$.

Como $A(X^{(0)})^T = A(0, 0, 0, 0)^T = (0, 0, 0, 0)^T$ entonces el sistema $DZ^{(0)} = (0, 0, 0, 0)^T$ tiene solución $Z^{(0)} = (0, 0, 0, 0)^T$.

- Debemos resolver $DW^{(0)} = B^T$. El sistema $DW^{(0)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(0)} = (-0.25, 0.5, 1, 2.5)^T$.

- Finalmente $X^{(1)} = X^{(0)} - Z^{(0)} + W^{(0)} = (-0.25, 0.5, 1, 2.5)^T$

- Segunda iteración.

- Debemos resolver $DZ^{(1)} = A(X^{(1)})^T$.

El sistema $DZ^{(1)} = A(-0.25, 0.5, 1, 2.5)^T$ tiene solución $Z^{(1)} = (-0.375, 0.3125, 0.25, 2.3125)^T$.

II.) Debemos resolver $DW^{(1)} = B^T$. El sistema $DW^{(1)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(1)} = (-0.25, 0.5, 1, 2.5)^T$.

III.) Finalmente $X^{(2)} = X^{(1)} - Z^{(1)} + W^{(1)} = (-0.125, 0.6875, 1.75, 2.6875)$

d.) Haga dos iteraciones a mano usando el método de Gauss-Seidel, con $X^{(0)} = (0, 0, 0, 0)$

Solución: La iteración es $X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)}$ donde $Z^{(k)}$ y $W^{(k)}$ son solución de los sistemas $L_AZ^{(k)} = AX^{(k)}$ y $L_AW^{(k)} = B$.

a.) Tenemos la aproximación inicial $X^{(0)} = (0, 0, 0, 0)$.

b.) La matriz L_A es $L_A = \begin{pmatrix} 4 & 0 & 0 & 0 \\ -1 & 4 & 0 & 0 \\ 0 & -1 & 4 & 0 \\ -1 & 0 & -1 & 4 \end{pmatrix}$

c.) **Primera iteración.**

I.) Debemos resolver $L_AZ^{(0)} = A(X^{(0)})^T$.

Como $A(X^{(0)})^T = A(0, 0, 0, 0)^T = (0, 0, 0, 0)^T$ entonces el sistema $L_AZ^{(0)} = (0, 0, 0, 0)^T$ tiene solución $Z^{(0)} = (0, 0, 0, 0)^T$.

II.) Debemos resolver $L_AW^{(0)} = B^T$. El sistema $L_AW^{(0)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(0)} = (-0.25, 0.4375, 1.109375, 2.714844)^T$.

III.) Finalmente $X^{(1)} = X^{(0)} - Z^{(0)} + W^{(0)} = (-0.25, 0.4375, 1.109375, 2.714844)^T$

d.) **Segunda iteración.**

I.) Debemos resolver $L_AZ^{(1)} = A(X^{(1)})^T$.

El sistema $L_AZ^{(1)} = A(-0.25, 0.4375, 1.109375, 2.714844)^T$ tiene solución $Z^{(1)} = (-0.359375, 0.1328125, 0.3544922, 2.498779)^T$.

II.) Debemos resolver $L_AW^{(1)} = B^T$. El sistema $L_AW^{(1)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(1)} = (-0.25, 0.4375, 1.109375, 2.714844)^T$.

III.) Finalmente $X^{(2)} = X^{(1)} - Z^{(1)} + W^{(1)} = (-0.140625, 0.7421875, 1.864258, 2.930908)$

e.) Haga dos iteraciones usando el método de SOR, con $X^{(0)} = (0, 0, 0, 0)$ y con $\omega = 1.05$

Solución: La iteración es $X^{(k+1)} = X^{(k)} - Z^{(k)} + W^{(k)}$ donde $Z^{(k)}$ y $W^{(k)}$ son solución de los sistemas $Q_\omega Z^{(k)} = AX^{(k)}$ y $Q_\omega W^{(k)} = B$ donde $Q_\omega = \left(\frac{1}{\omega}D - L_A\right)$.

a.) Tenemos la aproximación inicial $X^{(0)} = (0, 0, 0, 0)$.

$$\text{b.) } Q_\omega = \begin{pmatrix} 3.809524 & 0 & 0 & 0 \\ -1. & 3.809524 & 0 & 0 \\ 0. & -1. & 3.809524 & 0 \\ -1. & 0 & -1 & 3.809524 \end{pmatrix} \quad L_A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & -1 & 0 \end{pmatrix} \quad \text{y } D \text{ es la diagonal de } A.$$

c.) Primera iteración.

I.) Debemos resolver $Q_\omega Z^{(0)} = A(X^{(0)})^T$.

Como $A(X^{(0)})^T = A(0, 0, 0, 0)^T = (0, 0, 0, 0)^T$ entonces el sistema $Q_\omega Z^{(0)} = (0, 0, 0, 0)^T$ tiene solución $Z^{(0)} = (0, 0, 0, 0)^T$.

II.) Debemos resolver $Q_\omega W^{(0)} = B^T$. El sistema $Q_\omega W^{(0)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(0)} = (-0.2625, 0.4560938, 1.169725, 2.863146)^T$.

III.) Finalmente $X^{(1)} = X^{(0)} - Z^{(0)} + W^{(0)} = (-0.2625, 0.4560938, 1.169725, 2.863146)^T$

d.) Segunda iteración.

I.) Debemos resolver $Q_\omega Z^{(1)} = A(X^{(1)})^T$.

El sistema $Q_\omega Z^{(1)} = A(-0.2625, 0.4560938, 1.169725, 2.863146)^T$ tiene solución $Z^{(1)} = (-0.3953496, 0.1369727, 0.3928656, 2.767505)^T$.

II.) Debemos resolver $Q_\omega W^{(1)} = B^T$. El sistema $Q_\omega W^{(1)} = (-1, 2, 4, 10)^T$ tiene solución $W^{(1)} = (0.2625, 0.4560938, 1.169725, 2.863146)^T$.

III.) Finalmente $X^{(2)} = X^{(1)} - Z^{(1)} + W^{(1)} = (-0.1296504, 0.7752148, 1.946584, 2.958788)$

Ejercicio 5.4 Sea

$$A = \begin{pmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ -1 & 0 & -1 & 4 \end{pmatrix}$$

- a.) Verifique que la solución exacta es $X = (0, 1, 2, 3)^T$
- b.) Haga dos iteraciones a mano usando el método de Jacobi, con $X^{(0)} = (0, 0, 0, 0)$
- c.) Haga dos iteraciones a mano usando el método de Gauss-Seidel, con $X^{(0)} = (0, 0, 0, 0)$
- d.) Haga dos iteraciones usando el método de SOR, con $X^{(0)} = (0, 0, 0, 0)$ y con $\omega = 1.05$

5.5.1 Sistemas $m \times n$.

Consideremos el sistema $A_{m \times n}X = b$ con $m > n$. Si B está en el rango de A , entonces, por supuesto, el sistema tiene solución. Pero, en general, para un B arbitrario, lo mejor que podemos hacer es encontrar un vector X^* que *minimice* la norma euclíadiana:

$$X^* = \min_{X \in \mathbb{R}^n} \|AX - B\|_2^2$$

Al vector X^* le llamamos la “solución por mínimos cuadrados”.

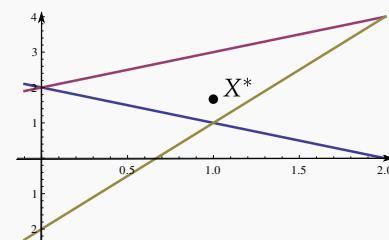
Ejemplo 5.10

Considere las tres rectas de ecuación $x + y = 2$, $x - y = -2$, y $3x - y = 2$. La representación gráfica la podemos ver en la figura de la derecha.

Claramente el sistema

$$\begin{cases} x + y = 2 \\ x - y = -2 \\ 3x - y = 2 \end{cases}$$

no tiene solución exacta, pero podemos buscar una solución X^* en el sentido de “mínimos cuadrados”. La solución buscada es $X^* = (1, 5/3)^T$



Del ejemplo anterior, se puede ver geométricamente que hay una solución X^* en el sentido de “mínimos cuadrados” si las tres rectas son “independientes” (no paralelas), es decir, si la matriz de coeficientes A tiene rango $\min(m, n)$. En este caso, la matriz $A^T A$ es simétrica y definida positiva y además la solución X^* existe y es única. De hecho, X^* es la solución del sistema

$$A^T A X = A^T B$$

■ **Descomposición QR.** La descomposición **QR** de una matriz $A_{m \times n}$ factoriza la matriz como $A = QR$ con $Q_{m \times m}$ ortogonal y R triangular superior. Esta descomposición se usa para resolver sistemas $m \times n$ con $m \geq n$ en el sentido de “mínimos cuadrados”. Para usar esta descomposición se usa la función **qr()** en la base de **R**.

Por ejemplo, para resolver el sistema,

$$\begin{cases} x + y &= 2 \\ x - y &= 0.1 \\ 3x - y &= 2 \end{cases}$$

en el sentido de “mínimos cuadrados”, se procede así

```
A = matrix(c(1, 1,
            1, -1,
            3, -1), nrow=3, byrow=TRUE)
b = c(2, 0.1, 2)
qr.solve(A,b)
# [1] 1.0000000 0.9666667
```

Veamos la aproximación de cerca: $\begin{cases} 1 + 0.9666667 &= 1.966667 \\ 1 - 0.9666667 &= 0.0333333 \\ 3 \cdot 1 - 0.9666667 &= 2.033333 \end{cases}$

Ejercicio 5.5 Resuelva, en el sentido de “mínimos cuadrados”, el sistema $AX = b$ donde $X = (x, y)^T$,

$$A = \begin{pmatrix} 0 & 1 \\ 0.06 & 1 \\ 0.14 & 1 \\ 0.25 & 1 \\ 0.31 & 1 \\ 0.47 & 1 \\ 0.60 & 1 \\ 0.70 & 1 \end{pmatrix} \quad \text{y} \quad B = \begin{pmatrix} 0 \\ 0.08 \\ 0.014 \\ 0.20 \\ 0.23 \\ 0.25 \\ 0.28 \\ 0.29 \end{pmatrix}$$

■



6 — Interpolación Polinomial

6.1 Introducción

La interpolación polinomial es la base de muchos tipos de integración numérica y tiene otras aplicaciones teóricas. En la práctica a menudo tenemos una tabla de datos $\{(x_i, y_i), i = 0, 1, 2, \dots, n\}$, obtenida por muestreo o experimentación. Suponemos que los datos corresponden a los valores de una función f desconocida (a veces es conocida, pero queremos cambiarla por una función más sencilla de calcular). El “ajuste de curvas” trata el problema de construir una función que aproxime muy bien estos datos (es decir, a f). Un caso particular de ajuste de curvas es la interpolación polinomial: En este caso se construye un polinomio $P(x)$ que pase por los puntos de la tabla.

La interpolación polinomial consiste en estimar $f(x^*)$ con $P(x^*)$ si x^* no está en la tabla pero se puede ubicar entre estos valores. Una situación típica se muestra en el siguiente ejemplo en el que tenemos datos que relacionan temperatura con el segundo coeficiente virial.¹

¹ El comportamiento de gases no ideales se describe a menudo con la *ecuación virial de estado*

$$\frac{PV}{RT} = 1 + \frac{B}{V} + \frac{C}{V^2} + \dots,$$

donde P es la presión, V el volumen molar del gas, T es la temperatura Kelvin y R es la constante de gas ideal. Los coeficientes $B = B(T)$, $C = C(T), \dots$ son el segundo y tercer coeficiente virial, respectivamente. En la práctica se usa la serie truncada

$$\frac{PV}{RT} \approx 1 + \frac{B}{V}$$

Ejemplo 6.1

Considere los siguientes datos para el nitrógeno (N_2):

$T(K)$	100	200	300	400	450	500	600
$B(cm^3/mol)$	-160	-35	-4.2	9.0	?	16.9	21.3

donde T es la temperatura y B es el segundo coeficiente virial. ¿Cuál es el segundo coeficiente virial a $450K$? Para responder la pregunta, usando interpolación polinomial, construimos un polinomio P que pase por los seis puntos de la tabla (ya veremos cómo), tal y como se muestra en la figura (6.1). Luego, el segundo coeficiente virial a $450K$ es aproximadamente $P(450) = 13.5cm^3/mol$.

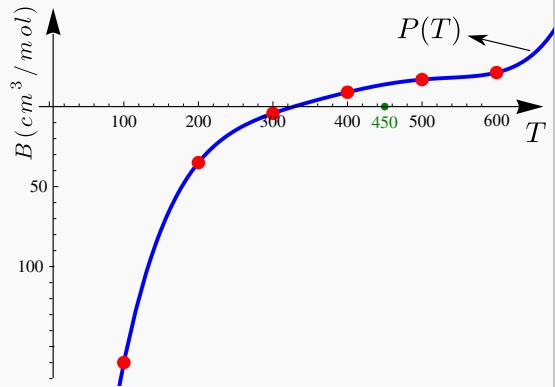


Figura 6.1: Polinomio interpolante

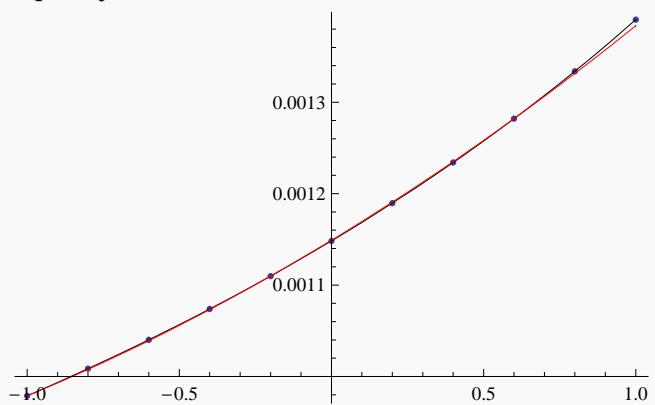
Ejemplo 6.2

Consideremos la función f definida por

$$f(x) = \int_5^\infty \frac{e^{-t}}{t-x} dt, \text{ con } -1 \leq x \leq 1$$

La integral que define a f es una integral no trivial (no se puede expresar en términos de funciones elementales). La tabla de la izquierda nos muestra algunos valores para f .

x	$f(x)$
-1	0.0009788055864607286
-0.6	0.0010401386051341144
-0.2	0.0011097929435687336
0	0.0011482955912753257
0.2	0.0011896108201581322
0.25	?
0.6	0.0012820294923443982
1.	0.0013903460525251596



Podemos usar un polinomio interpolante para interpolar $f(0.25)$.

En el mundillo del ajuste de curvas hay varias alternativas,

- Usar un polinomio interpolante. Es el método de propósito general más usado.

- Usar trazadores (splines). Estas son funciones polinomiales a trozos.
- Usar Polinomios trigonométricos en $[0, 2\pi]$. Son la elección natural cuando la función f es periódica de periodo 2π .
- Usar sumas exponenciales si conocemos que f presenta decaimiento exponencial conforme $x \rightarrow \infty$.
- Si los datos son aproximados (“datos experimentales”), lo conveniente sería usar *Mínimos Cuadrados*

Aquí vamos a tratar con interpolación polinomial, trazadores cúbicos y ...

6.2 Interpolación polinomial.

Un problema de interpolación polinomial se especifica como sigue: dados $n + 1$ pares $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, siendo todos los x_i 's distintos, y $y_i = f(x_i)$ para alguna función f ; encontrar un polinomio $P_n(x)$ de grado $\leq n$ tal que

$$P_n(x_i) = y_i, \quad i = 0, 1, 2, \dots, n$$

Teorema 6.1 (Polinomio interpolante).

Dados $n + 1$ puntos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ con $x_i \neq x_j$ si $i \neq j$; existe un único polinomio $P_n(x)$ de grado $\leq n$ tal que $P(x_i) = y_i \quad \forall i = 0, 1, \dots, n$

A $P_n(x)$ se le llama *polinomio interpolante*, a cada x_i le decimos *nodo de interpolación* y a cada y_i *valor interpolado*.

- El problema tiene solución única, es decir hay un único polinomio que satisface $P_n(x_i) = y_i$.
- No se requiere que los datos estén igualmente espaciados ni en algún orden en particular.
- Si f es un polinomio de grado $k \leq n$, el polinomio interpolante de f en $n + 1$ puntos coincide con f .
- El grado de P_n es $\leq n$ pues podría pasar, por ejemplo, que tres puntos estén sobre una recta y así el polinomio tendría grado cero o grado uno

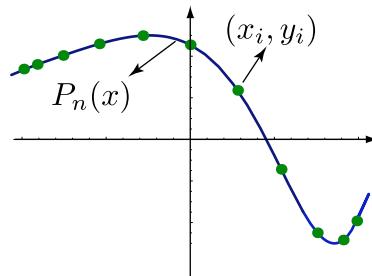


Figura 6.2: Polinomio interpolante.

Definición 6.1

Si de una función f conocemos los puntos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, con los x_i 's todos distintos (pero sin importar el orden), y si $A = \{x_0, x_1, \dots, x_n\}$ y $x^* \notin A$ pero $\min A < x^* < \max A$; entonces *interpolar* f en x^* con un subconjunto de $k + 1$ nodos de A consiste en calcular $P_k(x^*)$ donde P_k es el polinomio interpolante obtenido con un subconjunto de $k + 1$ nodos alrededor de x^* .

El polinomio interpolante es único, es decir, solo hay un polinomio que pasa por estos $n + 1$ puntos. Aquí vamos a ver cuatro maneras de calcular este polinomio interpolante: La forma de Lagrange del polinomio interpolante, la fórmula baricéntrica de Lagrange, la modificada de Lagrange y la forma de Newton del polinomio interpolante (método de diferencias divididas de Newton). Los cuatro métodos dan el mismo polinomio (aunque con diferente aspecto), y los cuatro métodos son importantes porque de ellos se hacen otras derivaciones teóricas.

6.3 Forma de Lagrange del polinomio interpolante.

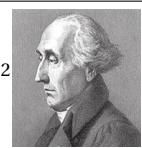
Lagrange² calculó el único polinomio interpolante de manera explícita: El polinomio $P_n(x)$ de grado $\leq n$ que pasa por los $n + 1$ puntos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ (con $x_i \neq x_j$ si $i \neq j$) es

$$P_n(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + \dots + y_n L_{n,n}(x)$$

donde $L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1}) \curvearrowright (x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1}) \curvearrowright (x_k - x_{k+1}) \cdots (x_k - x_n)}.$

Por ejemplo,

$$\begin{aligned} L_{n,0}(x) &= \frac{(x - x_1) \cdot (x - x_2) \cdots (x - x_n)}{(x_0 - x_1) \cdot (x_0 - x_2) \cdots (x_0 - x_n)}. \\ L_{n,1}(x) &= \frac{(x - x_0) \cdot (x - x_2) \cdots (x - x_n)}{(x_1 - x_0) \cdot (x_1 - x_2) \cdots (x_1 - x_n)}. \\ L_{n,3}(x) &= \frac{(x - x_0) \cdot (x - x_2) \cdot (x - x_4) \cdots (x - x_n)}{(x_3 - x_0) \cdot (x_3 - x_2) \cdot (x_3 - x_4) \cdots (x_3 - x_n)}. \\ &\vdots \quad \vdots \quad \vdots \\ L_{n,n}(x) &= \frac{(x - x_0) \cdot (x - x_1) \cdots (x - x_{n-1})}{(x_n - x_0) \cdot (x_n - x_1) \cdots (x_n - x_{n-1})}. \end{aligned}$$



² Joseph Louis Lagrange (1736-1813) fue uno de los más grandes matemáticos de su tiempo. Nació en Italia pero se nacionalizó Francés. Hizo grandes contribuciones en todos los campos de la matemática y también en mecánica. Su obra principal es la “Mécanique analytique” (1788). En esta obra de cuatro volúmenes, se ofrece el tratamiento más completo de la mecánica clásica desde Newton y sirvió de base para el desarrollo de la física matemática en el siglo XIX.

Ejemplo 6.3

Determine la forma de Lagrange polinomio interpolante de grado ≤ 2 (una recta o una parábola) que pasa por tres puntos $(0, 1), (1, 3), (2, 0)$.

Solución:

$$\begin{aligned} P_2(x) &= y_0 L_{2,0}(x) + y_1 L_{2,1}(x) + y_2 L_{2,2}(x) \\ &= 1 \cdot L_{2,0}(x) + 3 \cdot L_{2,1}(x) + 0 \cdot L_{2,2}(x) \\ &= 1 \cdot \frac{(x-1)(x-2)}{(0-1)(0-2)} + 3 \cdot \frac{(x-0)(x-2)}{(1-0)(1-2)} \end{aligned}$$

Ejemplo 6.4

De una función f , conocemos la información de la tabla que sigue. Interpolar $f(0.35)$ usando un polinomio interpolante $P_3(x)$ indicando la subtabla de datos que va a usar.

x	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
$f(x)$.3	.31	.32	.33	.34	.45	.46	.47

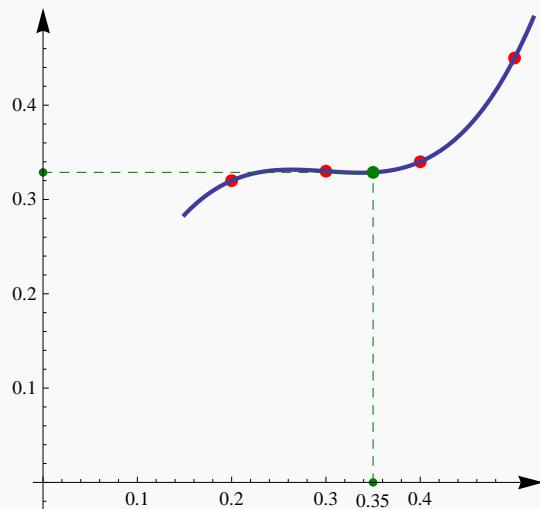
Solución: Como se requiere un polinomio interpolante $P_3(x)$, se necesita una subtabla de *cuatro* datos. Una opción es

x	0.2	0.3	0.4	0.5
$f(x)$	0.32	0.33	0.34	0.45

Si usamos la forma de Lagrange del polinomio interpolante, entonces

$$\begin{aligned} P_3(x) &= 0.32 \cdot \frac{(x-0.3)(x-0.4)(x-0.5)}{(0.2-0.3)(0.2-0.4)(0.2-0.5)} \\ &+ 0.33 \cdot \frac{(x-0.2)(x-0.4)(x-0.5)}{(0.3-0.2)(0.3-0.4)(0.3-0.5)} \\ &+ 0.34 \cdot \frac{(x-0.2)(x-0.3)(x-0.5)}{(0.4-0.2)(0.4-0.3)(0.4-0.5)} \\ &+ 0.45 \cdot \frac{(x-0.2)(x-0.3)(x-0.4)}{(0.5-0.2)(0.5-0.3)(0.5-0.4)} \end{aligned}$$

y entonces $f(0.35) \approx P_3(0.35) = 0.32875$.



Ejemplo 6.5 (Interpolación lineal. Fórmula de un solo punto para una recta).

Verifique que el polinomio interpolante de grado ≤ 1 que pasa por $(x_0, y_0), (x_1, y_1)$ es,

$$P_1(x) = m(x - x_1) + y_1 = \frac{(y_0 - y_1)}{(x_0 - x_1)}(x - x_1) + y_1$$

Solución: Usando la fórmula de Lagrange,

$$\begin{aligned} P_1(x) &= y_0 L_{n,0}(x) + y_1 L_{n,1}(x) \\ &= y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)}. \text{ Simplificando,} \\ &= \frac{(y_0 - y_1)}{(x_0 - x_1)}(x - x_1) + y_1 \end{aligned}$$

Ejemplo 6.6

En la tabla que sigue aparece las estadísticas de un curso con la cantidad de estudiantes en cada rango de notas.

Rango de Notas	30-40	40-50	50-60	60-70	70-80
Nº Estudiantes	35	48	70	40	22

Estime la cantidad de estudiantes con nota menor o igual a 55.

Solución: Para hacer la estimación necesitamos una tabla con las frecuencias acumuladas,

x≤	40	50	60	70	80
y	35	83	153	193	215

Ahora calculamos el polinomio interpolante,

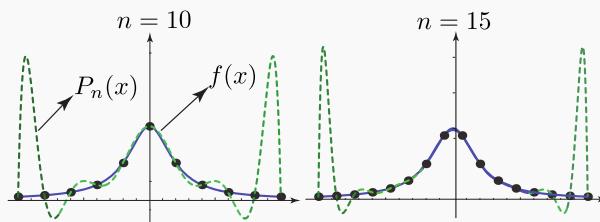
$$\begin{aligned} P_4(x) &= \frac{7(x - 80)(x - 70)(x - 60)(x - 50)}{48000} \\ &+ \frac{83(80 - x)(x - 70)(x - 60)(x - 40)}{60000} \\ &+ \frac{153(x - 80)(x - 70)(x - 50)(x - 40)}{40000} \\ &+ \frac{193(80 - x)(x - 60)(x - 50)(x - 40)}{60000} \\ &+ \frac{43(x - 70)(x - 60)(x - 50)(x - 40)}{48000} \end{aligned}$$

Así, la cantidad de estudiantes con nota menor o igual a 55 es aproximadamente $P_4(55) = 120$.

Ejemplo 6.7 (Nodos igualmente espaciados-fenómeno de Runge).

En general, el polinomio interpolante se podría ver afectado por el conjunto $\{x_0, \dots, x_n\}$ y por la función f .

Este ejemplo es algo extremo y es conocido como ‘fenómeno de Runge’; si $f(x) = \frac{1}{1+25x^2}$, el polinomio interpolante presenta problemas de convergencia si tomamos los x_i ’s igualmente espaciados en $[-1, 1]$, es decir si $x_i = -1 + i \cdot h$ con $h = 2/n$.



Observe que la interpolación se ve afectado hacia los extremos del intervalo no así en el centro; esto parece ser una tendencia general.

Si se puede escoger los nodos, una buena opción de ajuste se obtiene con nodos de Tchebychev³

Ejemplo 6.8 (Nodos de Tchebychev).

Si hay posibilidad de escoger los puntos de interpolación, en el intervalo $[-1, 1]$, la elección podría ser los nodos

$$\textcolor{red}{x}_i = \cos\left(\frac{2i+1}{2n+2}\pi\right),$$

conocidos como nodos de Tchebychev. A diferencia de lo que podría suceder con nodos igualmente espaciados, con estos nodos el polinomio interpolante ajusta bien si $f \in C^1[-1, 1]$.

Para un intervalo $[a, b]$ es válido hacer el cambio de variable $u = \frac{(b-a)(x-1)}{2} + b$ que mapea el intervalo $[-1, 1]$ en el intervalo $[a, b]$. En este caso, los nodos serían

$$u_i = \frac{(b-a)(\textcolor{red}{x}_i - 1)}{2+b} \quad \text{con} \quad \textcolor{red}{x}_i = \cos\left(\frac{2i+1}{2n+2}\pi\right).$$

³



Pafnuty Lvovich Tchebychev (1821 - 1894). El más prominente miembro de la escuela de matemáticas de St. Petersburg. Hizo investigaciones en Mecanismos, Teoría de la Aproximación de Funciones, Teoría de los Números, Teoría de Probabilidades y Teoría de Integración. Sin embargo escribió acerca de muchos otros temas: formas cuadráticas, construcción de mapas, cálculo geométrico de volúmenes, etc.

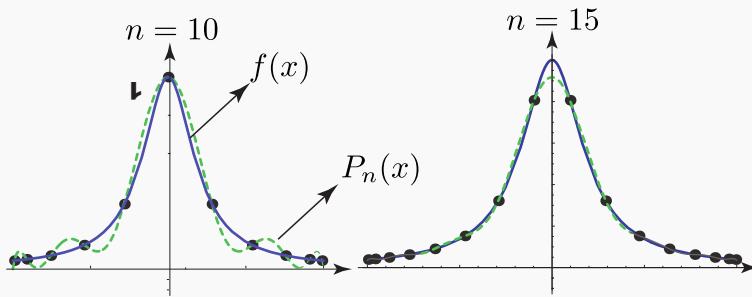


Figura 6.3: $P_n(x)$ con nodos $x_i = \cos((2i+1)/(2n+2)\pi)$.

Como se prueba más adelante, en este caso, si $x^* \in [a, b]$,

$$|f(x^*) - P_n(x^*)| \leq \frac{M}{(n+1)!} \frac{1}{2^n} \text{ si } |f^{(n+1)}(x)| \leq M \text{ para todo } x \in [a, b].$$

Implementación en R.

Recordemos que si tenemos los $n+1$ datos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ (con $x_i \neq x_j$ si $i \neq j$ (aunque los x_i no estén ordenados), entonces la forma de Lagrange del polinomio interpolante es

$$P_n(x) = \color{red}y_0\color{black} L_{n,0}(x) + \color{red}y_1\color{black} L_{n,1}(x) + \dots + \color{red}y_n\color{black} L_{n,n}(x)$$

$$\text{donde } L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}.$$

Para hacer una implementación vectorizada, recordemos que en los ejemplos 1.9 y 1.10 habíamos visto que si $\mathbf{x} = \mathbf{c}(\mathbf{x0}, \mathbf{x1}, \dots, \mathbf{xn})$ y si

$$\mathbf{X} = \mathbf{matrix}(\mathbf{rep}(\mathbf{x}, \mathbf{times=n}), \mathbf{n}, \mathbf{n}, \mathbf{byrow=T}) \text{ con } \mathbf{n} = \mathbf{length}(\mathbf{x}),$$

entonces, si hacemos $\mathbf{mN} = \mathbf{a} - \mathbf{X}$ y $\mathbf{diag}(\mathbf{mN}) = \mathbf{1}$, obtenemos

$$\mathbf{mN} = \begin{pmatrix} 1 & a - x_1 & a - x_2 & a - x_3 \\ a - x_0 & 1 & a - x_2 & a - x_3 \\ a - x_0 & a - x_1 & 1 & a - x_3 \\ a - x_0 & a - x_1 & a - x_2 & 1 \end{pmatrix}$$

y si hacemos $\mathbf{mD} = \mathbf{X} - \mathbf{t}(\mathbf{X})$ y $\mathbf{dia}(\mathbf{mD}) = \mathbf{1}$, obtenemos

$$\mathbf{mD} = \begin{pmatrix} 1 & x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ x_0 - x_1 & 1 & x_2 - x_1 & x_3 - x_1 \\ x_0 - x_2 & x_1 - x_2 & 1 & x_3 - x_2 \\ x_0 - x_3 & x_1 - x_3 & x_2 - x_3 & 1 \end{pmatrix}$$

Generalizando al caso $n \times n$, el numerador y el denominador de cada una de las funciones $L_{n,k}(a)$ se pueden obtener *aplicando* la función **prod** a las filas de **mN** y a las columnas de **mD**, por ejemplo

$$\text{Lnk}(a) = \text{prod}(\mathbf{N}[k, :]) / \text{prod}(\mathbf{D}[:, k])$$

■ Código R 6.1: Forma de Lagrange del polinomio interpolante

```
## Recibe los vectores de datos x y y "a" con min(x) < a < max(y).
## x = (x0,x1,x3,...,xn). Así x_k = x[k+1] y y_k = y[k+1]; k=0,1,...,n
## Devuelve P_n(a)
```

```
lagrange = function(x,y,a){
  n = length(x)
  if(a < min(x) || max(x) < a) stop("No está interpolando")
  X = matrix(rep(x, times=n), n, n, byrow=T)
  mN = a - X; diag(mN) = 1
  mD = X - t(X); diag(mD) = 1
  Lnk = apply(mN, 1, prod)/apply(mD, 2, prod)
  sum(y*Lnk)
}
```

Podemos usar el ejemplo 6.4 para hacer una prueba,

```
# ---
x = c( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
y = c(0.31, 0.32, 0.33, 0.34, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5)
lagrange(x[2:5],y[2:5], 0.35)
#[1] 0.32875
```

■ **Paquete “PolynomF” de R.** El polinomio interpolante se puede obtener con el paquete **PolynomF**. La función que calcula el polinomio interpolante es **poly.calc()**. Por ejemplo,

■ Código R 6.2: Función poly.calc del paquete PolynomF

```
# Instalar el paquete PolynomF
# install.packages("PolynomF")
require(PolynomF)
x = c( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0) # n+1 = 11
y = c(0.31, 0.32, 0.33, 0.34, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5)
# Polinomio de ajuste (polinomio interpolante en este caso)
datx = x[2:5]; daty = y[2:5]
polyAjuste = poly.calc(datx,daty)
polyAjuste
#-0.1 + 4.433333*x - 15*x^2 + 16.66667*x^3
plot(datx,daty, pch=19, cex=1, col = "red", asp=1) # Representación con puntos
curve(polyAjuste,add=T) # Curva de ajuste (polinomio interpolante) y puntos
#curve(polyAjuste,add=T, lty=3) #lty=3 puntos
```

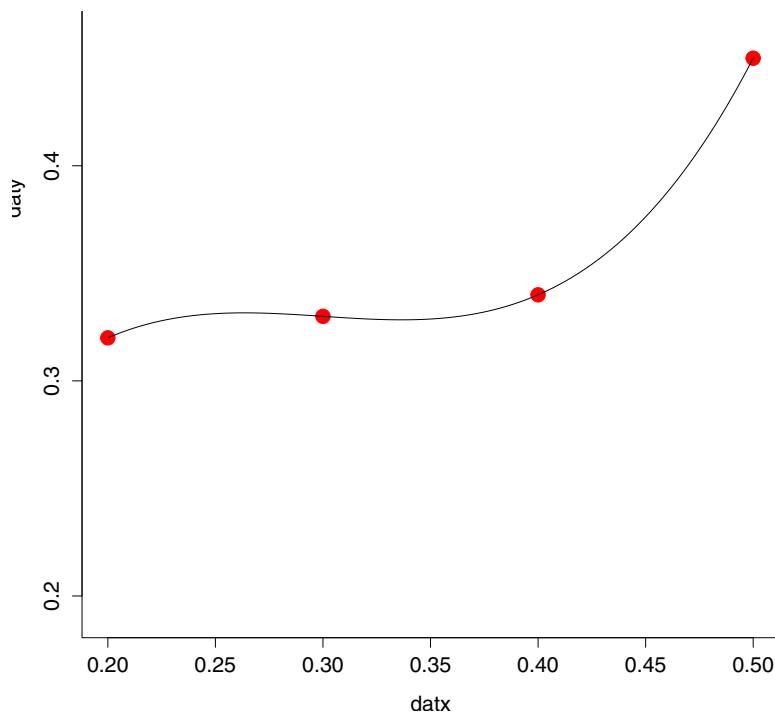


Figura 6.4: Ajuste de los datos con un polinomio interpolante

■ **Mejor ajuste.** Como mencionábamos antes, los polinomios interpolantes de grado alto “oscilan”. Por ejemplo,

■ Código R 6.3: “Oscilación” de $P_n(x)$

```
require(PolynomF)
xi=c(0,.5,1,2,3,4)
yi=c(0,.93,1,1.1,1.15,1.2)
polyAjuste = poly.calc(xi,yi)
#polyAjuste
plot(xi,yi, pch = 19, cex=1.5, col= "red")
curve(polyAjuste,add=T,lty=3, lwd=5)
```

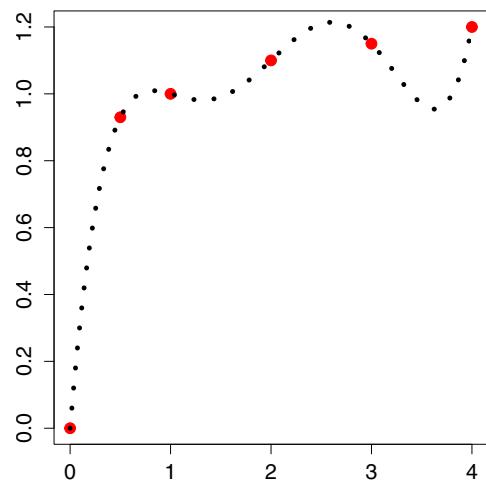


Figura 6.5: Ajuste de los datos con un polinomio interpolante de grado 5

6.4 Forma modificada y forma baricéntrica de Lagrange.

La forma de Lagrange del polinomio interpolante es atractiva para propósitos teóricos. Sin embargo se puede re-escribir en una forma que se vuelva eficiente para el cálculo computacional además de ser numéricamente mucho más estable (ver [2]). La forma modificada y la forma baricéntrica de Lagrange son útiles cuando queremos interpolar una función en todo un intervalo con un polinomio interpolante.

Supongamos que tenemos $n+1$ nodos distintos x_0, x_1, \dots, x_n . Sea $\ell(x) = \prod_{i=0}^n (x - x_i)$ es decir,

$$\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$$

Definimos los *pesos baricéntricos* como

$$\omega_k = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{1}{x_k - x_i}, \quad k = 0, 1, \dots, n.$$

Es decir,

$$\omega_k = \frac{1}{x_k - x_0} \cdot \frac{1}{x_k - x_1} \cdots \frac{1}{x_k - x_{k-1}} \cdot \frac{1}{x_k - x_{k+1}} \cdots \frac{1}{x_k - x_n}.$$

Ahora podemos definir la “forma modificada” y “forma baricéntrica” de Lagrange:

Definición 6.2

La *forma modificada* del polinomio de Lagrange se escribe como

$$P_n(x) = \ell(x) \sum_{j=0}^n \frac{\omega_j}{x - x_j} y_j \quad (6.1)$$

Definición 6.3

La *forma baricéntrica* del polinomio de Lagrange se escribe

$$P_n(x) = \begin{cases} = y_i & \text{si } x = x_i, \\ = \frac{\sum_{k=0}^n \frac{\omega_k}{x - x_k} y_k}{\sum_{k=0}^n \frac{\omega_k}{x - x_k}} & \text{si } x \neq x_i \end{cases} \quad (6.2)$$

Ejemplo 6.9

Consideremos la siguiente tabla de datos,

x	$f(x)$
0.2	3.2
0.3	3.3
0.4	3.4
0.5	4.5

Calcule la forma modificada y la forma baricéntrica de Lagrange e interpole con ambos polinomios, $f(0.35)$.

Solución: Primero calculamos $\ell(x) = (x - 0.2)(x - 0.3)(x - 0.4)(x - 0.5)$. Ahora, los pesos baricéntricos,

$$\begin{aligned} \omega_0 &= \frac{1}{0.2 - 0.3} \cdot \frac{1}{0.2 - 0.4} \cdot \frac{1}{0.2 - 0.5} = -166.667, \\ \omega_1 &= \frac{1}{0.3 - 0.2} \cdot \frac{1}{0.3 - 0.4} \cdot \frac{1}{0.3 - 0.5} = 500, \\ \omega_2 &= \frac{1}{0.4 - 0.2} \cdot \frac{1}{0.4 - 0.3} \cdot \frac{1}{0.4 - 0.5} = -500, \\ \omega_3 &= \frac{1}{0.5 - 0.2} \cdot \frac{1}{0.5 - 0.3} \cdot \frac{1}{0.5 - 0.4} = 166.667 \end{aligned}$$

Entonces, la *forma modificada* de Lagrange es,

$$P_3(x) = (x - 0.2)(x - 0.3)(x - 0.4)(x - 0.5) \left(-\frac{533.333}{x - 0.2} + \frac{1650.}{x - 0.3} - \frac{1700.}{x - 0.4} + \frac{750.}{x - 0.5} \right),$$

y la *forma baricéntrica* es,

$$P_3(x) = \frac{-\frac{533.333}{x - 0.2} + \frac{1650.}{x - 0.3} - \frac{1700.}{x - 0.4} + \frac{750.}{x - 0.5}}{-\frac{166.667}{x - 0.2} + \frac{500.}{x - 0.3} - \frac{500.}{x - 0.4} + \frac{166.667}{x - 0.5}}$$

En ambos casos, $f(0.35) \approx P_3(0.35) = 3.2875$.

Ejercicio 6.1 ■ Implementar la función `lagrangeModificada(x, y, a)`

■ **Paquete “barylag” en R.** La forma baricéntrica de Lagrange viene implementada como la función `barylag()` del paquete `pracma`. Por ejemplo,

```
require(pracma)
xi = c( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
yi = c(0.31, 0.32, 0.33, 0.34, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5)
# Interpolar con los 5 datos
xi[c(2:6)]; yi[c(2:6)]
# Interpolar en tres nodos: 0.35, 0.41, 0.55
barylag(xi[c(2:6)], yi[c(2:6)], c(0.35, 0.41, 0.55))
# [1] 0.2 0.3 0.4 0.5 0.6
# [1] 0.32 0.33 0.34 0.45 0.46
# barylag(xi[c(2:6)], yi[c(2:6)], c(0.35, 0.41, 0.55))
# [1] 0.3217187 0.3474487 0.4917187
```

5**Ejercicios**

6.1 Considere los cuatro puntos $(0, 1), (1, 2), (3, 0), (4, 4)$.

- Calcule el polinomio interpolante $P_3(x)$, en la forma de Lagrange.
- Verifique que efectivamente $P_4(x_i) = y_i$, es decir, $P_3(0) = 1, \text{etc.}$
- Interpolar $f(3.5)$.

6.2 Considere los cuatro puntos $(0, 1), (1, 2), (3, 0), (4, 4)$.

- Calcule el polinomio interpolante $P_3(x)$, en la forma de modificada.
- Calcule el polinomio interpolante $P_3(x)$, en la forma de Baricéntrica.
- Verifique que efectivamente $P_3(x_i) = y_i$, es decir, $P(0) = 1, \text{etc.}$
- Interpolar $f(3.5)$.

6.3 Consideremos la siguiente tabla de datos,

x	$f(x)$
0.2	1.2
0.3	5.3
0.4	9.4
0.5	10.5

Calcule la forma modificada y la forma baricéntrica de Lagrange e interpole $f(0.35)$. **Ayuda:** Estas fórmulas permiten reutilizar los cálculos!

6.4 Usando la forma de Lagrange del polinomio interpolante verifique que si $P(x)$ pasa por $(x_0, y_0), (x_1, y_1)$ entonces $P(x) = \frac{(y_0 - y_1)}{(x_0 - x_1)}(x - x_1) + y_1$. **Ayuda:** En algún momento de la simplificación debe sumar y restar

$y_1 x_1$.

6.5 Considere la función de Bessel $J_0(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \operatorname{sen} \theta) d\theta$. Tenemos la siguiente información,

x	$\pi J_0(x)$
0	3.59
0.2	3.11
0.4	3.08

- a.) Obtener la forma de Lagrange del polinomio interpolante.
- b.) Interpolar $J_0(0.25)$

6.6 Considere la siguiente tabla de salarios,

Salarios (\$)	0-1000	1000-2000	2000-3000	3000-4000
Frecuencia	9	30	35	42

Estimar la cantidad de personas con salario entre \$1000 y \$1500.

6.7 Interpolar $\cos(1.75)$ usando la tabla

x_i	$\cos(1 + 3x_i)$
0	0.540302
1/6	0.070737
1/3	-0.416147

Ayuda: La estimación que se obtiene con el polinomio interpolante es -0.17054 .

6.8 Considere la siguiente tabla de vapor para H_2O calentada a 200 MPa.

$v (m^3/kg)$	0.10377	0.11144	0.1254
$s (kJ/Kg \cdot K)$	6.4147	6.5453	6.7664

- a.) Use interpolación lineal para encontrar la entropía s para un volumen específico v de $0.108 m^3/kg$.
- b.) Use interpolación cuadrática para encontrar la entropía s para un volumen específico v de $0.108 m^3/kg$.

6.9 Usando la tabla del ejemplo (6.2), interpolar $f(0.25)$.

6.5 Forma de Newton para el polinomio interpolante.

La representación

$$P(x) = \color{red}{a_0} + \color{red}{a_1}(x - x_0) + \color{red}{a_2}(x - x_0)(x - x_1) + \cdots + \color{red}{a_n}(x - x_0) \cdots (x - \color{blue}{x_{n-1}}),$$

para el polinomio interpolante que pasa por los $n+1$ puntos $(x_0, y_0), \dots, (x_n, y_n)$, es conocida como *la representación de Newton* del polinomio interpolante.

6.6 Diferencias Divididas de Newton.

La manera más conocida para calcular la representación de Newton del polinomio interpolante, está basada en el método de *diferencias divididas*. Una gran ventaja sobre la forma clásica del método de Lagrange es que podemos agregar más nodos a la tabla de datos y obtener el polinomio interpolante sin tener que recalcular todo. Comparado con la forma modificada de Lagrange, no hay ganancia y más bien esta última forma es más estable. Aún así, el método de diferencias divididas tiene aplicaciones adicionales en otros contextos.

Podemos calcular los a_i 's usando el hecho de que $P(x_i) = y_i$,

$$\left\{ \begin{array}{lcl} P(x_0) = y_0 & = & a_0 \\ P(x_1) = y_1 & = & a_0 + a_1(x_1 - x_0) \\ P(x_2) = y_2 & = & a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) \\ & \vdots & \end{array} \right. \Rightarrow \begin{array}{l} a_0 = y_0, \\ a_1 = \frac{y_1 - y_0}{x_1 - x_0}, \\ a_2 = \frac{y_2 - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \end{array}$$

Si $y_k = f(x_k)$, la fórmula anterior nos muestra que cada a_k depende de x_0, x_1, \dots, x_k . Desde muchos años atrás se usa la notación $a_k = f[x_0, x_1, \dots, x_k]$ para significar esta dependencia.

Al símbolo $f[x_0, x_1, \dots, x_n]$ se le llama *diferencia divida* de f . Usando esta nueva notación tendríamos que la forma de Newton del polinomio interpolante es

$$\begin{aligned} P(x) &= \color{red}{f[x_0]} + \color{red}{f[x_0, x_1]}(x - x_0) + \color{red}{f[x_0, x_1, x_2]}(x - x_0)(x - x_1) \\ &\quad + \cdots + \color{red}{f[x_0, \dots, x_n]}(x - x_0) \cdots (x - \color{blue}{x_{n-1}}), \end{aligned}$$

donde $\color{red}{f[x_0]} = y_0$ y $\color{red}{f[x_0, \dots, x_i]}$ es el coeficiente principal de la forma de Newton del polinomio que interpola la función f en los nodos $\color{blue}{x_0}, \color{blue}{x_1}, \dots, \color{blue}{x_i}$.

Ejemplo 6.10 (Interpolación lineal).

El polinomio interpolante de grado ≤ 1 que pasa por $(x_0, y_0), (x_1, y_1)$ es

$$P_1(x) = f[x_0] + f[x_0, x_1](x - x_0) \text{ donde } f[x_0, x_1] = \frac{(y_0 - y_1)}{(x_0 - x_1)} \text{ y } f[x_0] = y_0$$

Si consideramos al coeficiente $f[x_0, x_1, \dots, x_n]$ como una función de $n+1$ variables, entonces esta función es *simétrica*, es decir, permutar las variables de cualquier manera no afecta el valor de la función. Esto es así porque el polinomio que interpola los puntos $\{(x_i, y_i)\}_{i=0, \dots, n}$ es único, por lo tanto sin importar el orden en que vengan los puntos, el coeficiente principal siempre es $a_n = f[x_0, x_1, \dots, x_n]$.

¿Qué es $f[x_k, x_{k+1}, \dots, x_{k+j}]$? Es el coeficiente principal de la forma de Newton del polinomio que interpola una función f en los nodos $x_k, x_{k+1}, \dots, x_{k+j}$. Por ejemplo, si tenemos $n+1$ datos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, el polinomio que interpola $(x_3, y_3), (x_4, y_4)$ sería

$$P_1(x) = y_3 + f[x_3, x_4](x - x_3).$$

El nombre “*diferencia dividida*” viene del hecho de que cada $f[x_k, x_{k+1}, \dots, x_{k+j}]$ se puede expresar como un cociente de diferencias.

Teorema 6.2

La diferencia dividida $f[x_k, x_{k+1}, \dots, x_{k+j}]$ satisface la ecuación

$$f[x_k, x_{k+1}, \dots, x_{k+j}] = \frac{f[x_{k+1}, x_{k+2}, \dots, x_{k+j}] - f[x_k, x_{k+1}, \dots, x_{k+j-1}]}{x_{k+j} - x_k} \quad (6.3)$$

Ejemplo 6.11

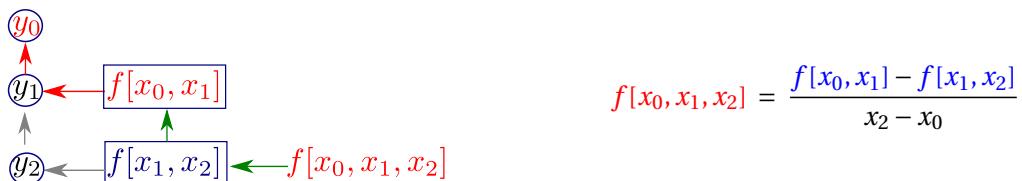
El teorema (6.2) indica que cada diferencia dividida se puede calcular en términos de otras “diferencias” previamente calculadas. Los ejemplos que siguen son casos particulares para mostrar cómo se aplica el teorema.

$$\begin{aligned}
 f[x_i, x_j] &= \frac{y_i - y_j}{x_i - x_j} \\
 f[\textcolor{red}{x_0}, x_1, \textcolor{red}{x_2}] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{\textcolor{red}{x_2} - \textcolor{red}{x_0}} \\
 f[\textcolor{red}{x_1}, x_2, \textcolor{red}{x_3}] &= \frac{f[x_2, x_3] - f[x_1, x_2]}{\textcolor{red}{x_3} - \textcolor{red}{x_1}} \\
 &\vdots \\
 f[\textcolor{red}{x_0}, x_1, x_2, \textcolor{red}{x_3}] &= \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{\textcolor{red}{x_3} - \textcolor{red}{x_0}} \\
 f[\textcolor{red}{x_1}, x_2, x_3, \textcolor{red}{x_4}] &= \frac{f[x_2, x_3, x_4] - f[x_1, x_2, x_3]}{\textcolor{red}{x_4} - \textcolor{red}{x_1}} \\
 &\vdots \\
 f[\textcolor{red}{x_0}, x_1, \dots, \textcolor{red}{x_k}] &= \frac{f[x_1, x_2, \dots, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{\textcolor{red}{x_k} - \textcolor{red}{x_0}}
 \end{aligned}$$

Este esquema recursivo se puede arreglar en forma matricial como sigue,

x_0	y_0				
x_1	y_1	$f[\textcolor{blue}{x_0}, x_1]$			
x_2	y_2	$f[x_1, x_2]$	$f[\textcolor{blue}{x_0}, \textcolor{blue}{x_1}, x_2]$		
x_3	y_3	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	$f[\textcolor{blue}{x_0}, x_1, x_2, x_3]$	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

En general, para calcular $f[x_0], f[x_0, x_1], f[x_0, x_1, x_2], \dots, f[x_0, \dots, x_n]$, debemos calcular una matriz en la que las nuevas columnas se construyen con los datos de la columna anterior.



La misma matriz se puede usar para calcular la forma de Newton para subconjuntos de datos: En el arreglo que sigue, la diagonal principal (en rojo) corresponde a los coeficientes del polinomio que interpola los datos $(x_0, y_0), \dots, (x_n, y_n)$. La diagonal en azul corresponde a los coeficientes del polinomio que interpola los datos $(x_1, y_1), \dots, (x_n, y_n)$.

$$\begin{array}{ll}
 y_0 & \\
 y_1 & f[x_0, x_1] \\
 y_2 & f[x_1, x_2] \quad f[x_0, x_1, x_2] \\
 y_3 & f[x_2, x_3] \quad f[x_1, x_2, x_3] \\
 \vdots & \vdots \quad \ddots \\
 y_n & f[x_{n-1}, x_n] \quad f[x_{n-2}, x_{n-1}, x_n] \quad \cdots \quad f[x_1, \dots, x_n] \quad f[x_0, x_1, \dots, x_n]
 \end{array}$$

Por ejemplo, para calcular el polinomio que interpola los datos $(x_3, y_3), \dots, (x_6, y_6)$ se usa la (sub)matriz,

$$\begin{array}{ll}
 y_3 & \\
 y_4 & f[x_3, x_4] \\
 y_5 & f[x_4, x_5] \quad f[x_3, x_4, x_5] \\
 y_6 & f[x_5, x_6] \quad f[x_1, x_2, x_3] \quad f[x_3, x_4, x_5, x_6]
 \end{array}$$

La diagonal principal (en rojo) corresponde a los coeficientes del polinomio que interpola estos cuatro datos.



■ Programa en Internet (applet Java):

<http://www.tec-digital.itcr.ac.cr/revistamatematica/cursos-linea/NumericoApplets/DifDivNewton.htm>

Ejemplo 6.12

Usando diferencias divididas, calcular el polinomio interpolante para los datos $(-1, 2), (1, 1), (2, 2), (3, -2)$ y el polinomio interpolante para los datos $(1, 1), (2, 2), (3, -2)$.

Solución: Primero construimos la matriz de diferencias divididas usando todos los datos. En rojo están los coeficientes del polinomio que interpola todos los datos y en azul los coeficientes del polinomio que interpola los datos $(1, 1), (2, 2), (3, -2)$.

$$\begin{array}{lllll}
 x_0 & y_0 & & & 2 \\
 x_1 & y_1 & f[x_0, x_1] & & 1 & -1/2 \\
 x_2 & y_2 & f[x_1, x_2] & f[x_0, x_1, x_2] & 2 & 1 & 1/2 \\
 x_3 & y_3 & f[x_2, x_3] & f[x_1, x_2, x_3] & f[x_0, x_1, x_2, x_3] & -2 & -4 & -5/2 & -3/4
 \end{array} \longrightarrow$$

El polinomio interpolante, en la forma de Newton, para todos los datos es

$$P(x) = 2 - 1/2(x+1) + 1/2(x+1)(x-1) - 3/4(x+1)(x-1)(x-2)$$

El polinomio interpolante, en la forma de Newton, para los datos $(1, 1), (2, 2), (3, -2)$ es

$$P(x) = 1 + 1 \cdot (x - 1) + -5/2(x - 1)(x - 2)$$

Ejemplo 6.13

De una función f , conocemos la información de la tabla (6.6). Interpolar $f(0.35)$ usando un polinomio interpolante $P_3(x)$. Primero que todo, escriba la tabla de datos que va a usar.

x	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
$f(x)$	3	3.1	3.2	3.3	3.4	4.5	4.6	4.7

Solución: Como se requiere un polinomio interpolante $P_3(x)$, se necesita una tabla de cuatro datos. Una opción es

x	0.2	0.3	0.4	0.5
$f(x)$	3.2	3.3	3.4	4.5

Si usamos la forma de Newton del polinomio interpolante, entonces

$\begin{array}{r} 3.2 \\ 3.3 \quad 1 \\ 3.4 \quad 1 \quad 0 \\ 4.5 \quad 11 \quad 50 \quad 166.6\bar{6} \end{array}$	$\begin{aligned} P_3(x) &= 3.2 + 1 \cdot (x - 0.2) \\ &+ 0 \cdot (x - 0.2)(x - 0.3) \\ &+ 166.6\bar{6} \cdot (x - 0.2)(x - 0.3)(x - 0.4) \end{aligned}$
--	---

Por tanto $f(0.35) \approx P_3(0.35) = 3.2875$

Implementación en R.

La forma de Newton del polinomio interpolante es

$$P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \dots + f[x_0, \dots, x_n](x - x_0) \dots (x - x_{n-1}),$$

donde los coeficientes están en la diagonal de la matriz de diferencias divididas,

$$\begin{array}{ll} y_0 = f[x_0] & \\ y_1 & f[x_0, x_1] \\ y_2 & f[x_1, x_2] \quad f[x_0, x_1, x_2] \\ y_3 & f[x_2, x_3] \quad f[x_1, x_2, x_3] \\ \vdots & \vdots \quad \ddots \\ y_n & f[x_{n-1}, x_n] \quad f[x_{n-2}, x_{n-1}, x_n] \quad \dots \quad f[x_0, x_1, \dots, x_n] \end{array}$$

Para la implementación de la fórmula de diferencias divididas de Newton es conveniente reescribir el polinomio como

$$P(x) = C_{0,0} + C_{1,1}(x - x_0) + C_{2,2}(x - x_0)(x - x_1) \cdots + C_{n,n}(x - x_0) \cdots (x - x_{n-1})$$

y la matriz de diferencias divididas como

$$\begin{array}{ccccccccc} y_0 & = & C_{0,0} & & & & & & \\ y_1 & & & C_{1,1} & & & & & \\ y_2 & & & & C_{2,2} & & & & \\ y_3 & & & & & C_{3,2} & & & \\ \vdots & & & & \vdots & & \ddots & & \\ y_n & & & & & C_{n,1} & C_{n,2} & \cdots & C_{n,n} \end{array}$$

Para el cálculo de los $C_{i,i}$'s usamos la fórmula recursiva:

$$\begin{aligned} C_{i,0} &= y_i, \quad i = 0, 1, \dots, n \\ C_{i,j} &= \frac{C_{i,j-1} - C_{i-1,j-1}}{x_i - x_{i-j}}, \quad j = 1, 2, \dots, n, \quad i = j, j+1, \dots, n \end{aligned}$$

La fórmula de diferencias divididas dice que la columna j se calcula con la columna $j-1$ pero en cada columna solo nos interesa la posición diagonal $C_{j,j}$ y las entradas $C_{j+1,j}, \dots, C_{n,j}$.

Algoritmo 6.1: Forma de Newton del polinomio interpolante (sin vectorizar)

Datos: $\{(x_i, y_i)\}_{i=0,1,\dots,n}$ con los x_i 's distintos y x^*

Salida: $P_n(x^*)$

```

1 for  $i = 1$  ton do
2    $C_{i,0} = y_i$ 
3 ;
4 for  $j = 1$  ton do
5   for  $i = j$  ton do
6      $C_{i,j} = \frac{C_{i,j-1} - C_{i-1,j-1}}{x_i - x_{i-j}}$ 
7  $sum = C_0;$ 
8  $factor = 1;$ 
9 for  $j = 1$  ton do
10    $factor = factor \cdot (x^* - x_{j-1});$ 
11    $sum = sum + C_{j,j} \cdot factor;$ 
12 return sum

```

■ **Algoritmo semi-vectorizado.** Para vectorizar el algoritmo solo hay que observar que el cálculo de las C_{ij} se puede obtener como una resta de vectores “desfazados”.,

Supongamos que A es la matriz de diferencias divididas; ahora observemos los numeradores y los denominadores de la columna j , de la fórmula recursiva, de manera esquemática sería:

$$C_{i,0} = y_i, \quad i = 0, 1, 2, \dots, n, \text{ es decir, } A[, 1] = y_i$$

Numeradores

$$\begin{aligned} i = n & \quad C_{n,j-1} - C_{n-1,j-1} \\ \vdots & \quad \vdots \\ i = j+1 & \quad C_{j+1,j-1} - C_{j,j-1} \\ i = j & \quad C_{j,j-1} - C_{j-1,j-1} \\ C_{i,j} & = \frac{C_{i,j-1} - C_{i-1,j-1}}{x_i - x_{i-j}} = \frac{A[j:n, j-1] - A[(j-1):(n-1), j-1]}{x[j:n] - x[0:(n-j)]} \end{aligned}$$

Denominadores

$$\begin{aligned} i = j & \quad x_j - x_0 \\ i = j+1 & \quad x_{j+1} - x_1 \\ \vdots & \quad \vdots \\ i = n & \quad x_n - x_{n-j} \end{aligned}$$

En el código ajustamos los índices, porque A y x inician en 1 no en 0 .

■ Código R 6.4: Matriz de diferencias divididas e interpolación

```
newtonInterpolacion = function(x, y, a) {
  n = length(x)
  A = matrix(rep(NA, times = n^2), nrow = n, ncol = n)
  A[,1] = y
  for (k in 2:n) {
    A[k:n, k] = (A[k:n, k-1] - A[(k-1):(n-1), k-1]) / (x[k:n] - x[1:(n-k+1)])
  }
  # Imprimir matriz de diferencias divididas
  print(A)
  # Evaluar
  smds = rep(NA, length = n)
  smds[1] = 1 # x = x[1], ..., x[n] pues n = length(x)
  for (k in 2:n) {
    smds[k] = (a - x[k-1])*smds[k-1] # hasta x[n-1]
  }
  return(sum(diag(A)*smds) )
}

##---- pruebas -----
```

```

x = c( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
y = c(0.31, 0.32, 0.33, 0.34, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5)
newtonInterpolacion(x[2:5], y[2:5], 0.35)
# --- Matriz de diferencias divididas
# [,1] [,2]      [,3]      [,4]
#[1,] 0.32    NA      NA      NA
#[2,] 0.33    0.1     NA      NA
#[3,] 0.34    0.1   -2.775558e-16  NA
#[4,] 0.45    1.1   5.000000e+00 16.66667
# --- Interpolación
#[1] 0.32875

```

6.7 Forma de Lagrange vs Forma de Newton.

Usualmente se reserva la forma de Lagrange del polinomio interpolante para trabajo teórico y diferencias divididas de Newton para cálculos. La realidad es que la *forma modificada de Lagrange* es tan eficiente como diferencias divididas de Newton en cuanto a costo computacional y además es numéricamente mucho más estable. Hay varias ventajas que hacen de esta forma modificada de Lagrange, el método a escoger cuando de interpolación polinomial se trata ([8], [9]).

Para mostrar la inestabilidad del polinomio interpolante obtenido con diferencias divididas versus el obtenido con la forma modificada de Lagrange, consideramos la función de Runge $f(x) = 1/(1 + 25x^2)$ en $[-1, 1]$. Para un buen ajuste, usamos 52 nodos de Tchebyshov. En la figura 6.6, se muestra la gráfica de f junto con la gráfica del polinomio interpolante obtenido con diferencias divididas ($PN(x)$) y del polinomio interpolante obtenido con la forma modificada de Lagrange ($PML(x)$). Usando la aritmética usual de la máquina, se nota inestabilidad de $PN(x)$ en las cercanías de $x = -1$. En la figura 6.7, se muestra el error relativo de la aproximación a f con cada polinomio en $[-1, -0.9]$. $EPN(x)$ corresponde al error relativo entre f y la forma de Newton del polinomio interpolante y $EPML(x)$ corresponde al error relativo entre f y la forma modificada de Lagrange.

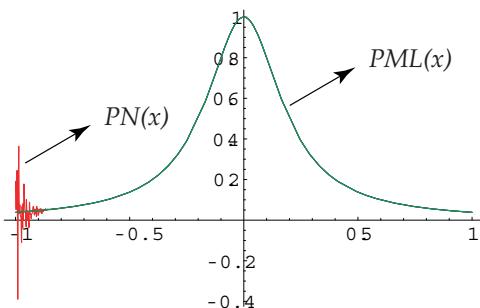


Figura 6.6: Interpolación. Diferencias divididas vs forma modificada de Lagrange

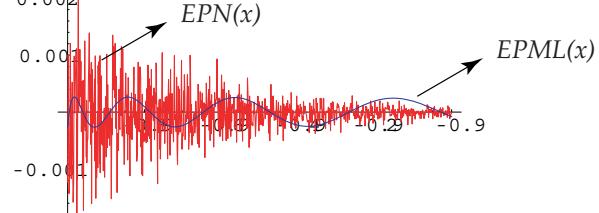


Figura 6.7: Error relativo.

6.8 Estimación del error.

La estimación del error, cuando interpolamos con un polinomio interpolante, es de interés práctico en varias áreas, por ejemplo en el desarrollo de métodos de aproximación en ecuaciones diferenciales ordinarias y en ecuaciones diferenciales en derivadas parciales.

Una estimación del error se puede obtener si conocemos alguna información acerca de la función f y sus derivadas. Sea $f \in C^{n+1}[a, b]$ y $P_n(x)$ el polinomio de interpolación de f en $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, con $x_i \in [a, b]$. Entonces, usando polinomios de Taylor podemos establecer la siguiente fórmula para el error

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n)$$

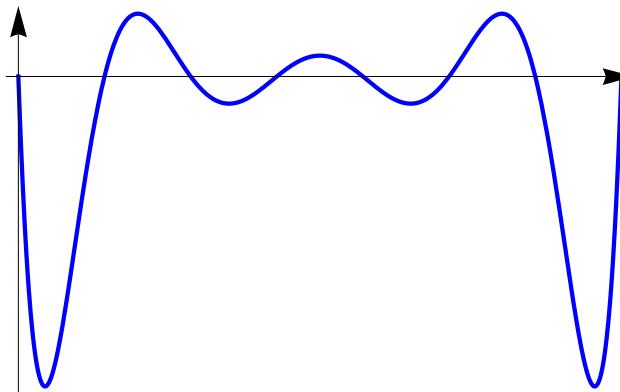
donde $a < \xi(x) < b$ y $x \in [a, b]$. Aquí, la expresión “ $\xi(x)$ ” significa que ξ no es una constante fija, sino que varía según el valor que tome x .

Para efectos prácticos, a y b son el mínimo y el máximo del conjunto $\{x_0, x_1, \dots, x_n\}$. Si M_n es el el máximo absoluto de la función $|f^{(n+1)}|$ en $[a, b]$, es decir, $|f^{(n+1)}(x)| \leq M_n$ para todo $x \in [a, b]$, entonces podemos obtener una estimación del error $f(x) - P_n(x)$ con la desigualdad,

$$|f(x) - P_n(x)| \leq \frac{M_n}{(n+1)!} |(x - x_0)(x - x_1) \cdots (x - x_n)|; \quad x \in [a, b]. \quad (6.4)$$

Observe que un polinomio interpolante de grado alto no garantiza una mejora en el error: Si usamos más puntos (posiblemente más cercanos entre ellos) se puede esperar que el producto $\prod_i (x - x_i)$ se haga más pequeño con n , pero todavía debería pasar que la derivada de orden $n+1$ no crezca más rápido que $(n+1)!$ y esto parece no ser la regla⁴.

Si los nodos son igualmente espaciados, y suponiendo que tenemos n y M_n fijos, la estimación del error depende de la función $\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$. La forma general de esta función se muestra en la figura que sigue,



⁴

Georg Faber (1912) demostró que para cada juego de nodos, existe un función continua para la cual los polinomios interpolantes no convergen uniformemente a f y también, para cada función continua existe un juego de nodos donde los polinomios interpolantes si convergen de manera uniforme. Aún en este último caso, los nodos no siempre fáciles de obtener.

Figura 6.8: $\ell(x) = (x - x_0) \cdots (x - x_7)$ con 7 nodos igualmente espaciados.

Esto sugiere que en el caso de nodos igualmente espaciados (excepto $n = 1$), el error es más pequeño si x está hacia el centro y empeora en los extremos.

La desigualdad (6.4) sería suficiente para estimar el error al interpolar en un valor x , pero nos interesa también una estimación que nos sirva para todo $x \in [x_0, x_n]$.

6.9 Error en interpolación lineal.

Si tenemos dos puntos $(x_0, y_0), (x_1, y_1)$ con $x_0 < x_1$, el error es $|f(x) - P_1(x)| = \frac{(x - x_0)(x - x_1)}{2} f''(\xi(x))$. ¿Cuál es el error máximo si x está entre x_0 y x_1 y si f'' permanece acotada en $[x_0, x_1]$?

Si $|f''(x)| \leq M_2$ en $[x_0, x_1]$, entonces

$$|f(x) - P_1(x)| \leq \frac{M_2}{2!} |(x - x_0)(x - x_1)|.$$

El error máximo depende del máximo valor de la función

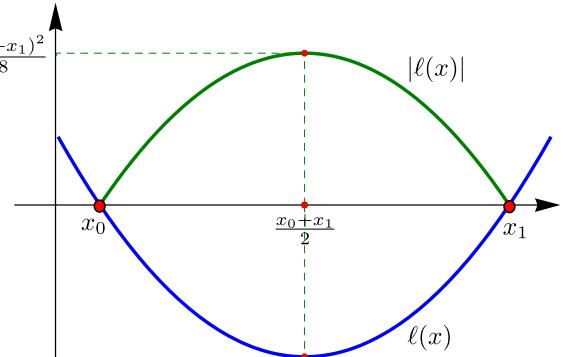
$\left| \frac{(x - x_0)(x - x_1)}{2} \right|$ en el intervalo $[x_0, x_1]$.

Como $\ell(x) = \frac{(x - x_0)(x - x_1)}{2}$ es una parábola cónica hacia arriba (figura 6.9), es negativa si $x \in [x_0, x_1]$, por lo tanto el máximo en valor absoluto lo alcanza en $x = \frac{x_0 + x_1}{2}$, y es

$$\frac{(x_1 - x_0)^2}{8}.$$

∴ Si se usa interpolación lineal, el error general está acotado por

$$|f(x) - P_1(x)| \leq M_2 \frac{(x_1 - x_0)^2}{8}.$$

**Figura 6.9:** $\ell(x) = (x - x_0)(x - x_1)$ y $|\ell(x)|$

Ejemplo 6.14

Si tabulamos la función $\sin x$ para $x_0 = 0, x_1 = 0.002, x_2 = 0.004, \dots$ entonces el error general al *interpolar linealmente* es

$$|\sin x - P_1(x)| \leq 1 \cdot \left| \frac{(0.002)^2}{8} \right| = 0.5 \times 10^{-6},$$

pues $|\sin x| \leq 1 \quad \forall x$ (Aquí suponemos que el polinomio se evalúa de manera exacta). Esto nos dice que la función $\sin x$ es apropiada para interpolación lineal.

Si deseamos más precisión en un caso particular, podemos usar la fórmula (6.4). Si por ejemplo $x = 0.003$, entonces $|\sin(0.003) - P_1(0.003)| \leq \frac{\sin(0.004)}{2} |(0.003 - 0.002)(0.003 - 0.004)| \approx -1.99 \times 10^{-9}$ pues el máximo absoluto de $|\sin x|$ en el intervalo $[0.002, 0.004]$ es $\sin(0.004)$.

6.10 Error en interpolación cuadrática

Si interpolamos con tres puntos ($n = 2$) igualmente espaciados $x_0, x_1 = x + h$ y $x_2 = x_0 + 2h$; entonces si $x \in [x_0, x_2]$ y si $|f'''(x)| \leq M_3$ en $[a, b]$, la estimación general del error es,

$$\begin{aligned}|f(x) - P_2(x)| &\leq \frac{M_3}{3!} |(x - x_0)(x - x_1)(x - x_2)| \\ &\leq \frac{M_3}{6} |(x - x_0)(x - x_0 - h)(x - x_0 - 2h)|\end{aligned}$$

Para obtener el máximo absoluto de la función $\ell(x) = (x - x_0)(x - x_1)(x - x_2)$ calculamos sus puntos críticos: $\ell'(x) = 3x^2 + x(-6h - 6x_0) + 6hx_0 + 3x_0^2 + 2h^2$, los ceros de esta cuadrática son

$$r_1 = \frac{1}{3}(3h + \sqrt{3}h + 3x_0) \quad y \quad r_2 = \frac{1}{3}(3h - \sqrt{3}h + 3x_0).$$

Como $\ell(x)$ se anula en x_0 y x_2 , el máximo absoluto de $|\ell(x)|$ es $\max\{|\ell(r_1)|, |\ell(r_2)|\} = \frac{2h^3}{3\sqrt{3}}$.

∴ El error general al interpolar con tres *puntos igualmente espaciados* es $|f(x) - P_2(x)| \leq \frac{M_3 h^3}{9\sqrt{3}}, \quad x \in [x_0, x_2]$.

Ejemplo 6.15

Si tabulamos la función $\sin x$ para $x_0 = 0, x_1 = 0.01, x_2 = 0.02, \dots$ entonces el error general al *interpolar con un polinomio de grado dos* es

$$|\sin x - P_2(x)| \leq \frac{1 \cdot (0.01)^3}{9\sqrt{3}} \approx 6.415 \times 10^{-8},$$

pues $|\sin x| \leq 1 \quad \forall x$.

6.11 Error en interpolación cúbica

Si tenemos cuatro puntos *igualmente espaciados* $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)$ con $x_0 < x_1 < x_2 < x_3$, una estimación del error es

$$|f(x) - P_3(x)| \leq \frac{M_4}{4!} |(x - x_0)(x - x_1)(x - x_2)(x - x_3)|, \quad \text{con } |f^{(4)}(x)| \leq M_4 \text{ en } [x_0, x_3].$$

De nuevo, dados n y M_4 fijos, la estimación del error general depende del máximo absoluto del polinomio $|\ell(x)| = |(x - x_0)(x - x_1)(x - x_2)(x - x_3)|$. Como $x_i = x_0 + i \cdot h$, $i = 1, 2, 3$;

$$\begin{aligned}\ell(x) &= (x - x_0)(x - x_1)(x - x_2)(x - x_3) \\ &= (x - x_0)(x - x_0 - h)(x - x_0 - 2h)(x - x_0 - 3h)\end{aligned}$$

$$\ell'(x) = 2(2x - 3h - 2x_0)(x^2 + (-3h - 2x_0) + h^2 + 3hx_0 + x_0^2)$$

Los puntos críticos son $r_1 = 0.5(3h + 2x_0)$, $r_2 = 0.5(3h - \sqrt{5}h + 2x_0)$ y $r_3 = 0.5(3h + \sqrt{5}h + 2x_0)$. Como $\ell(x)$ se anula en x_0 y x_3 , entonces el máximo absoluto de $|\ell(x)|$ es $\max\{|\ell(r_1)|, |\ell(r_2)|, |\ell(r_3)|\} = \left\{\frac{9h^4}{16}, h^4\right\} = h^4$. Finalmente,

∴ El error general al interpolar con cuatro puntos igualmente espaciados es $|f(x) - P_3(x)| \leq \frac{M_4 h^4}{24}$, $x \in [x_0, x_3]$.

∴ Si solo interpolamos valores $x \in [x_1, x_2]$, el máximo absoluto de $|\ell(x)|$ en este intervalo se alcanza en el punto medio $x = (x_1 + x_3)/2 = 0.5(3h + 2x_0)$ y es $\frac{9h^4}{16}$. En este caso la estimación del error general es

$$|f(x) - P_3(x)| \leq \frac{3M_4 h^4}{128}, \quad x \in [x_1, x_2].$$

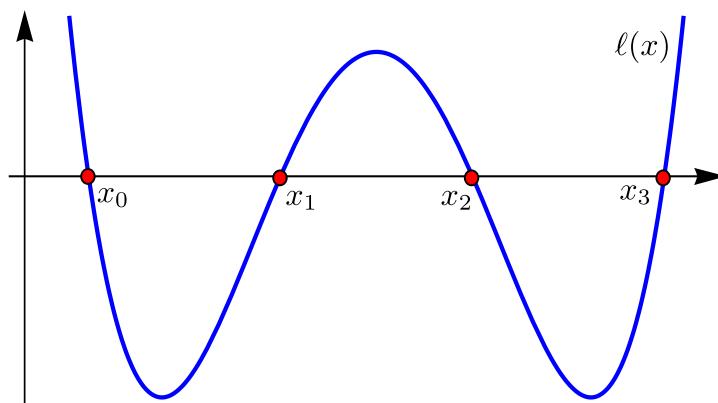


Figura 6.10: $\ell(x) = (x - x_0)(x - x_1)(x - x_2)(x - x_3)$

Ejemplo 6.16

Si tabulamos la función $\sin x$ para $x_0 = 0$, $x_1 = 0.05$, $x_2 = 0.10$, $x_3 = 0.15$, ... entonces el error al interpolar con P_3 entre x_1 y x_2 es

$$|\sin x - P_3(x)| \leq 1 \cdot \frac{3}{128} (0.05)^4 \approx 1.46 \times 10^{-7},$$

pues $|\sin x| \leq 1 \forall x$ (Aquí suponemos que el polinomio se evalúa de manera exacta).

6.12 Error con interpolación con polinomios de grado n .

Si interpolamos sobre *puntos igualmente espaciados* $x_i = x_0 + i \cdot h$, $i = 0, 1, \dots, n$; y si h es pequeño entonces $f^{(n+1)}(\xi(x))$ en general no se espera que varíe gran cosa. El comportamiento del error es entonces principalmente determinado por $\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$.

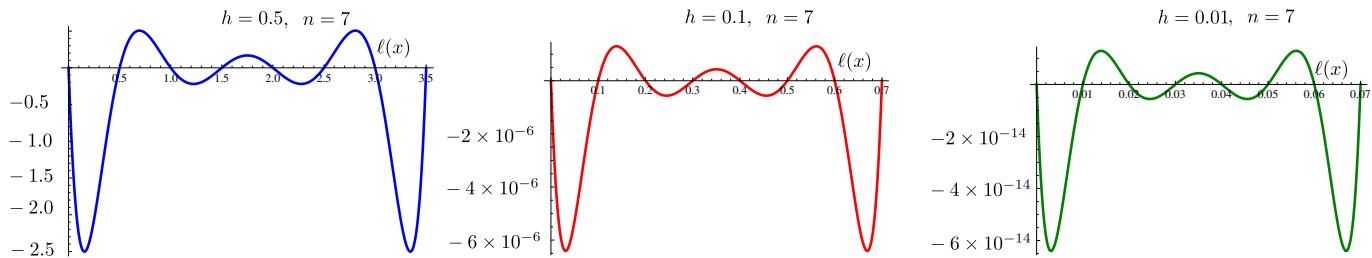


Figura 6.11: $\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$ con $n = 7$.

Pero las oscilaciones de $\ell(x)$ se hacen más violentas si n crece,

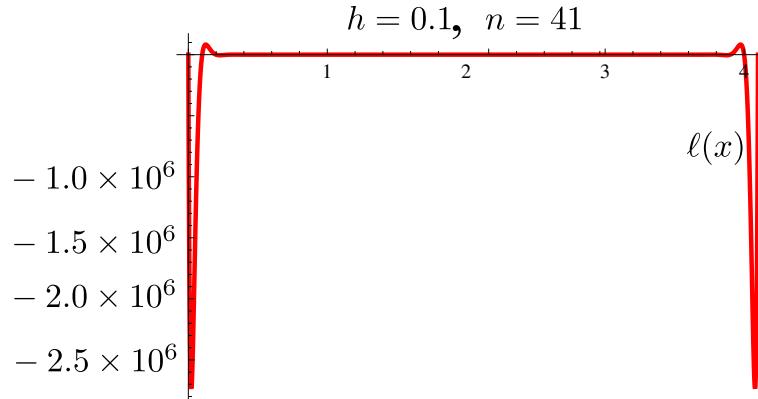


Figura 6.12: $\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$ con $n = 41$.

Sin embargo, la sucesión de polinomios interpolantes $\{P_n(x)\}$ podría converger a f (sin importar si los nodos son o no igualmente espaciados); esto depende del comportamiento de la derivada k -ésima de f : La sucesión $\{P_n(x)\}$ converge a f uniformemente en $[a, b]$ (que contiene a los nodos) si

$$\lim_{k \rightarrow \infty} \frac{(b-a)^k}{k!} M_k = 0$$

y esto sucede si f es *analítica* en una región suficientemente grande, en el plano complejo, que contenga a $[a, b]$ ([1, pág 84]).

6.13 Otros casos.

Si la función f y sus derivadas son conocidas, se puede hacer una estimación del error con el máximo absoluto.

Ejemplo 6.17

Sea $f(x) = \frac{1}{2} e^{(x-1)/2}$. Usando la fórmula de error, estime el error que se cometería al interpolar $f(1)$ con el polinomio interpolante obtenido de la tabla

x	$f(x)$
0.7	0.43
0.8	0.45
1.1	0.53
1.2	0.55

Solución: Son cuatro datos (no igualmente espaciados), $n + 1 = 4$. Luego, la fórmula para estimar el error es

$$|f(1) - P_3(1)| = \left| \frac{f^{(4)}(\xi)}{4!} (1-0.7)(1-0.8)(1-1.1)(1-1.2) \right| \leq \left| \frac{M}{4!} (1-0.7)(1-0.8)(1-1.1)(1-1.2) \right|$$

donde M es el máximo absoluto de $|f^{(4)}(x)| = |\frac{1}{32} e^{\frac{x-1}{2}}|$, en $[0.7, 1.2]$

Cálculo de M

Puntos críticos: La ecuación $f^{(5)}(x) = \frac{1}{64} e^{\frac{x-1}{2}} = 0$ no tiene solución, así que no hay puntos críticos.

Comparación: $M = \max\{|f^{(4)}(0.7)|, |f^{(4)}(1.2)|\} = 0.0345366\dots$

Finalmente, la estimación del error es $|f(1) - P_3(1)| \leq \left| \frac{M}{4!} (1-0.7)(1-0.8)(1-1.1)(1-1.2) \right| = 1.72683 \times 10^{-6}$

6

6.10 Sea $f(x) = x^2 \ln x - x^2$. Supongamos que $P(x)$ es el polinomio interpolante de f obtenido con los datos $(1, -1), (2, -1.2), (3, 0.88)$. Estime el error cometido al aproximar $f(2.71)$ con $P(2.71)$.

6.11 Sea $f(x) = \ln(4x^2 + 2)$. Usando la fórmula de error, estime el error que se cometería al interpolar $f(1.22)$ con el polinomio interpolante obtenido de la tabla

x	$f(x)$
0.5	1.09861
1.1	1.92279
1.2	2.04898
1.3	2.1702

6.12 Considere la tabla de datos

x	e^x
0	1
0.5	$e^{0.5}$
1	e

Estime el error cometido al aproximar $e^{0.6}$ con el polinomio de interpolación correspondiente, en el intervalo $[0, 1]$.

6.13 Sea $f(x) = \cos(3x + 1)$. Supongamos que $P(x)$ es el polinomio interpolante de f obtenido con los datos $(0., 0.54), (0.5, -0.80), (1., -0.65)$. Estime el error cometido al estimar $f(0.71)$ con $P(0.71)$.

6.14 Considere la tabla de datos

x_i	$\cos(1 + 3x_i)$
0	0.540302
1/6	0.070737
1/3	-0.416147

Estime el error cometido al interpolar $\cos(1.75)$ con el polinomio de interpolación obtenido con la tabla anterior, en el intervalo $[0, 1/3]$. **Ayuda:** Observe que en este caso, $x \neq 1.75$!

6.15 Sea $f(x) = \frac{1}{2}(\cos x + \operatorname{sen} x)$. Considere el conjunto de puntos $\{(x_i, f(x_i))\}_{i=0,1,2,3}$ con $x_i = i \cdot \pi/2$. Estime el error *general* cometido al aproximar $f(3\pi/4)$ con $P_3(3\pi/4)$.

6.16 Sea $f(x) = \frac{x^6}{84} - \frac{3 \cos(2x)}{8}$. Considere el conjunto de puntos $\{(x_i, f(x_i))\}_{i=0,1,2,3}$ con $x_i = i \cdot 0.2$. Estime el error *general* cometido al aproximar $f(0.65)$ con $P_3(0.65)$.

6.17 Aproximar $F(0.25; 1, 2)$ (función gamma, ver ejemplo 6.18) usando interpolación lineal, cuadrática y cúbica.

6.14 (*) Interpolación Iterada de Neville

Si no tenemos información acerca de las derivadas de una función no podemos usar la fórmula para el cálculo del error. Entonces, ¿cuál es el grado del polinomio de interpolación más adecuado para interpolar un valor? Para responder esta pregunta podemos usar el *algoritmo de Neville*, este método interpola un valor particular con polinomios de grado cada vez más alto (iniciando en grado cero) hasta que los valores sucesivos están suficientemente cercanos. Luego por inspección podemos decidirnos por un valor en particular.

Usemos la siguiente notación: $P_{0,1}$ es el polinomio interpolante que pasa por $(x_0, y_0), (x_1, y_1)$; $P_{0,1,2}$ es el polinomio

interpolante que pasa por $(x_0, y_0), (x_1, y_1), (x_2, y_2)$; $P_{1,2,3,4}$ es el polinomio interpolante que pasa por $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$; etc. Como no tenemos información acerca de las derivadas de f , el criterio para estimar el error es empírica e implícita: Nos quedamos con la estimación que presente 'menos variación'.

Consideremos la siguiente tabla de datos,

x	1.2	1.3	1.4	1.5	1.6
$f(x)$	0.7651977	0.6200860	0.4554022	0.2818186	0.1103623

Para interpolar en $x = 1.35$ tenemos varias opciones y combinaciones, con tres nodos, con cuatro nodos, etc. Usando nuestra notación, algunos resultados son $P_{0,1,2}(1.35) = 0.5401905$; $P_{123}(1.35) = 0.5388565$; $P_{0123}(1.35) = 0.5395235$; $P_{1234}(1.35) = 0.5395457$; $P_{01234}(1.35) = 0.5395318$. La menor variación la encontramos con $P_{0123}(1.35) = 0.5395235$; $P_{1234}(1.35) = 0.5395457$ y $P_{01234}(1.35) = 0.5395318$ y de estos tres, los más cercanos son $P_{0123}(1.35) = 0.5395235$ y $P_{01234}(1.35) = 0.5395318$. En este caso parece lo mejor quedarnos con la aproximación $P_{01234}(1.35) = 0.5395318$ ya que toma en cuenta toda la tabla.

El problema en el análisis anterior es la gran cantidad de polinomios que se deben evaluar, el algoritmo de Neville precisamente automatiza esta tarea usando cálculos anteriores para obtener el nuevo cálculo. El algoritmo de Neville no calcula $P(x)$ sino que evalúa varios polinomios interpolantes de Lagrange en un valor dado.

Sea $Q_{i,j}$ el polinomio interpolante que pasa por $(x_{i-j}, y_{i-j}), \dots, (x_i, y_i)$, es decir,

$Q_{i,j} = P_{i-j, i-j+1, i-j+2, \dots, i-1, i}$ es el polinomio interpolante (en la forma de Lagrange) que pasa por los nodos $(x_{i-j}, y_{i-j}), (x_{i-j+1}, y_{i-j+1}), \dots, (x_i, y_i)$, $0 \leq j \leq i$. Por ejemplo,

$Q_{0,0} = P_0$ pasa por (x_0, y_0) , es decir, $P_0(x_0) = y_0$.

$Q_{4,0} = P_4$ pasa por (x_4, y_4) , es decir, $P_4(x_4) = y_4$.

$Q_{5,2} = P_{3,2,1}$ pasa por $(x_3, y_3), (x_4, y_4), (x_5, y_5)$

$Q_{4,4} = P_{0,1,2,3,4}$ pasa por $(x_0, y_0), (x_1, y_1), \dots, (x_4, y_4)$

Con esta definición de $Q_{i,j}$ se tiene la siguiente relación recursiva,

$$Q_{i,j}(x) = \frac{(x - x_{i-j})Q_{i,j-1}(x) - (x - x_i)Q_{i-1,j-1}(x)}{x_i - x_{j-1}} \quad (6.5)$$

Aplicando esta relación para $i = 1, 2, \dots, n; j = 1, 2, \dots, i$ se logra calcular varios polinomios interpolantes de Lagrange en un valor x , como se muestra en la siguiente tabla (para el caso de 5 nodos)

x_0	$Q_{0,0} = y_0$					
x_1	$Q_{1,0} = y_1$	$Q_{1,1} = P_{0,1}$				
x_2	$Q_{2,0} = y_2$	$Q_{2,1} = P_{1,2}$	$Q_{2,2} = P_{0,1,2}$			
x_3	$Q_{3,0} = y_3$	$Q_{3,1} = P_{2,3}$	$Q_{3,2} = P_{1,2,3}$	$Q_{3,3} = P_{0,1,2,3}$		
x_4	$Q_{4,0} = y_4$	$Q_{4,1} = P_{3,4}$	$Q_{4,2} = P_{2,3,4}$	$Q_{4,3} = P_{1,2,3,4}$	$Q_{4,4} = P_{0,1,2,3,4}$	



■ Programa en Internet:

<http://www.tec-digital.itcr.ac.cr/revistamatematica/cursos-linea/NumericoApplets/Neville.htm>

Ejemplo 6.18

La distribución gamma se define como $F(x; \beta, \alpha) = \int_0^{x/\beta} \frac{u^{\alpha-1} e^{-u}}{\Gamma(\alpha)} du$

Supogamos que tenemos la siguiente tabla de datos, obtenida con $\beta = 1$ y $\alpha = 2$.

x	$F(x; 1, 2)$
$x_0 = 0$	0.0
$x_1 = 0.1$	0.00467884
$x_2 = 0.2$	0.01752309
$x_3 = 0.3$	0.03693631
$x_4 = 0.4$	0.06155193

Si queremos estimar F en 0.25 debemos usar polinomios que al menos pasen por x_2 y x_3 . Por ejemplo $P_{0,1,2,3}$, $P_{1,2,3,4}$, etc.

Aplicando el algoritmo de Neville en $x = 0.25$, obtenemos la tabla (redondeado a 7 cifras decimales),

x_0	P_0				
0	0.0				
x_1	P_1	$P_{0,1}$			
0.1	0.0046679	0.0116697			
x_2	P_2	$P_{1,2}$	$P_{0,1,2}$		
0.2	0.0175231	0.0239507	0.0270209		
x_3	P_3	$P_{2,3}$	$P_{1,2,3}$	$P_{0,1,2,3}$	
0.3	0.0369363	0.0272297	0.0264099	0.0265118	
x_4	P_4	$P_{3,4}$	$P_{2,3,4}$	$P_{1,2,3,4}$	$P_{0,1,2,3,4}$
0.4	0.0615519	0.0246285	0.0265794	0.0264947	0.0265011

La menor variación la tenemos entre $P_{1,2,3,4}$ y $P_{0,1,2,3,4}(0.25)$. Como $F(0.25; 1, 2) = 0.026499021\dots$, la mejor aproximación en realidad es $P_{1,2,3,4}$, pero en la práctica, por supuesto, tomamos decisiones sin esta información.

Algoritmo

El algoritmo es muy parecido al método de diferencias divididas de Newton, escribimos la primera columna de la matriz Q (las y_i 's) y luego completamos la matriz con la relación recursiva (6.5).

Figura 6.13: Algoritmo de Neville

Datos: Valor a interpolar x y los nodos $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Salida: Matriz Q

```

1 for  $i = 0, \dots, n$  do
2    $Q_{i,0} = y_i$ 
3 for  $i = 1, \dots, n$  do
4   for  $j = 1, \dots, i$  do
5      $Q_{i,j}(x) = \frac{(x - x_{i-j})Q_{i,j-1}(x) - (x - x_i)Q_{i-1,j-1}(x)}{x_i - x_{j-1}}$ 
6 return Matriz  $Q$ 
```

■ **Implementación en R.** El algoritmo de Neville viene implementada como la función `neville()` del paquete `pracma`. Por ejemplo,

```

require(pracma)
xi = c( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
yi = c(0.31, 0.32, 0.33, 0.34, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5)

neville(xi[c(2:5)], yi[c(2:5)], 0.35)
neville(xi[c(2:6)], yi[c(2:6)], 0.35)

# [1] 0.32875

# [1] 0.3217187
```

7

Ejercicios

6.18 Complete la fila 6 en la tabla

x_0	$Q_{0,0} = P_0$						
x_1	$Q_{1,0} = P_1$	$Q_{1,1} = P_{0,1}$					
x_2	$Q_{2,0} = P_2$	$Q_{2,1} = P_{1,2}$	$Q_{2,2} = P_{0,1,2}$				
x_3	$Q_{3,0} = P_3$	$Q_{3,1} = P_{2,3}$	$Q_{3,2} = P_{1,2,3}$	$Q_{3,3} = P_{0,1,2,3}$			
x_4	$Q_{4,0} = P_4$	$Q_{4,1} = P_{3,4}$	$Q_{4,2} = P_{2,3,4}$	$Q_{4,3} = P_{1,2,3,4}$	$Q_{4,4} = P_{0,1,2,3,4}$		
x_5	$Q_{5,0} = P_5$		

6.19 Use el algoritmo de Neville para aproximar $F(0.25; 1, 2)$ usando nuestro criterio empírico para obtener una “mejor aproximación”.

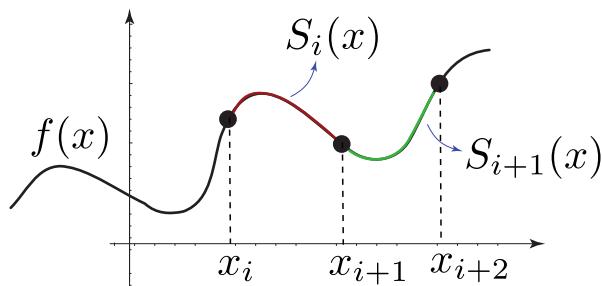
6.20 Supongamos que x_0, x_1, \dots, x_n son nodos distintos de un intervalo finito $[a, b]$. Sea $P_n(x)$ el polinomio interpolante obtenido con los datos $\{(x_i, f(x_i))\}_{i=0,1,\dots,n}$. Si $|f^{(n+1)}(x)| \leq M$ para x en $[a, b]$, muestre que si $x^* \in [a, b]$,

$$|f(x^*) - P_n(x^*)| \leq \frac{M}{(n+1)!} (b-a)^{n+1}$$

6.15 Trazadores Cúbicos (Cubic Splines).

Un trazador (spline) es una banda de hule delgada y flexible que se usa para dibujar curvas suaves a través de un conjunto de puntos. Los trazadores cúbicos (cubic splines) *naturales* se utilizan para crear una función que interpola un conjunto de puntos de datos. Esta función consiste en una unión de polinomios cúbicos, uno para cada intervalo, y está construido para ser una función con derivada primera y segunda continuas. El ‘spline’ cúbico natural también tiene su segunda derivada igual a cero en la coordenada x del primer punto y el último punto de la tabla de datos.

Supongamos que tenemos $n + 1$ puntos $(x_0, y_0), \dots, (x_n, y_n)$ con $x_0 < x_1 < \dots < x_n$. En vez de interpolar f con un solo polinomio que pase por todos estos puntos, interpolamos la función f en cada subintervalo $[x_k, x_{k+1}]$ con un polinomio cúbico (en realidad de grado ≤ 3) $S_k(x)$ de tal manera que el polinomio cúbico (o trazador cúbico) $S_i(x)$ en $[x_i, x_{i+1}]$ y el trazador cúbico $S_{i+1}(x)$ en $[x_{i+1}, x_{i+2}]$, coincidan en x_{i+1} y que también sus derivadas primera y segunda coincidan en este punto. Cada trazador cúbico coincide con f en los extremos de cada intervalo.



Definición 6.4 (Trazador Cúbico).

Un *trazador cúbico* S es una función a trozos que interpola a f en los $n + 1$ puntos $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ (con $a = x_0 < x_1 < \dots < x_n = b$). S es definida de la siguiente manera,

$$S(x) = \begin{cases} S_0(x) & \text{si } x \in [x_0, x_1], \\ S_1(x) & \text{si } x \in [x_1, x_2], \\ \vdots & \vdots \\ S_{n-1}(x) & \text{si } x \in [x_{n-1}, x_n], \end{cases}$$

Donde,

(a). $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$ para $i = 0, 1, \dots, n - 1$

■ Interpolación

(b). $S(x_i) = y_i$, $i = 0, 1, \dots, n$. Para efectos prácticos, $S_j(x_j) = y_j$, $j = 0, 1, \dots, n - 1$ y $S_{n-1}(x_{n-1}) = y_{n-1}$ y $S_{n-1}(x_n) = y_n$. El siguiente ítem asegura que $S_j(x_{j+1}) = y_{j+1}$.

■ Continuidad

(c). $S_i(x_{i+1}) = S_{i+1}(x_{i+1})$ para $i = 0, 1, \dots, n - 2$

■ Suavidad

(d). $S'_i(x_{i+1}) = S'_{i+1}(x_{i+1})$ para $i = 0, 1, \dots, n - 2$

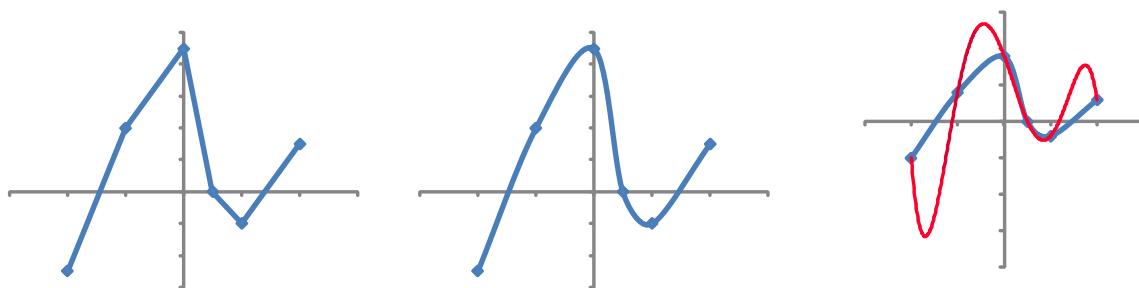
(e). $S''_i(x_{i+1}) = S''_{i+1}(x_{i+1})$ para $i = 0, 1, \dots, n - 2$

(f). Se satisface una de las dos condiciones que siguen,

(i). $S''(x_0) = S''(x_n) = 0$ (**frontera libre o natural**)

(ii). $S'(x_0) = f'(x_0)$ y $S'(x_n) = f'(x_n)$ (**frontera sujetada**)

Una aplicación directa de los trazadores cúbicos es la de “suavizar curvas”. Tanto en Excel como en *Calc* de OpenOffice o LibreOffice, en las gráficas de dispersión, los pares ordenados (x_i, y_i) se pueden unir con segmentos, con trazadores cúbicos o con el polinomio interpolante (también hay otras opciones, según el modelo o tendencia que se esté aplicando). En la gráfica de la figura que sigue se muestra un conjunto de datos unidos por segmentos, unidos por trazadores cúbicos y unidos por el polinomio interpolante.



(a) Líneas

(b) Trazadores cúbicos

(c) Trazadores y polinomio interpolante

Algunas curvas presentan “picos” así que se construye un trazador para cada curva entre cada dos picos. El tratamiento de picos requiere usualmente un trazador con frontera sujeta.

El proceso de construcción del trazador cúbico consiste en determinar cada polinomio cúbico $S_j(x)$, es decir, buscar sus coeficientes a_i , b_i , c_i y d_i . La definición nos da las condiciones que se deben cumplir. De estas condiciones podemos obtener un sistema de ecuaciones $4n \times 4n$, donde las incógnitas son todos los coeficientes a_i , b_i , c_i y d_i , $i = 0, 1, \dots, n - 1$. Lo que obtenemos es un trazador cúbico único.

Ejemplo 6.19

Determinar el trazador cúbico (frontera libre) para la siguiente tabla,

x_i	$y_i = \cos(3x_i^2) \ln(x_i^3 + 1)$
$x_0 = 0$	0
$x_1 = 0.75$	-0.0409838
$x_2 = 1.5$	1.31799

Solución: El trazador es, $S(x) = \begin{cases} S_0(x) = a_0 + b_0(x - x_0) + c_0(x - x_0)^2 + d_0(x - x_0)^3 & \text{si } x \in [x_0, x_1], \\ S_1(x) = a_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 & \text{si } x \in [x_1, x_2]. \end{cases}$

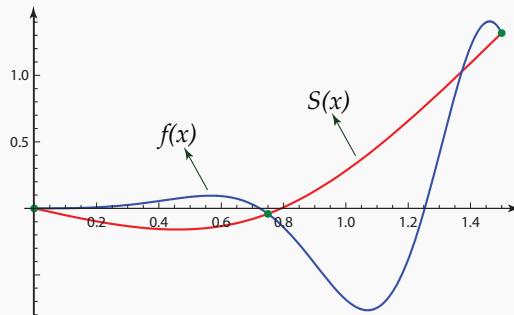
Hay que determinar los coeficientes de S_0 y S_1 resolviendo el sistema 8×8 ,

$$\left\{ \begin{array}{lcl} S_0(x_0) & = & y_0 \\ S_1(x_1) & = & y_1 \\ S_1(x_2) & = & y_2 \\ S_0(x_1) & = & S_1(x_1) \\ S'_0(x_1) & = & S'_1(x_1) \\ S''_0(x_1) & = & S''_1(x_1) \\ S'''_0(x_0) & = & 0 \\ S'''_1(x_2) & = & 0 \end{array} \right. \iff \left\{ \begin{array}{lcl} a_0 & = & 0 \\ a_1 & = & -0.0409838 \\ a_1 + 0.75b_1 + 0.5625c_1 + 0.421875d_1 & = & 1.31799 \\ a_0 + 0.75b_0 + 0.5625c_0 + 0.421875d_0 & = & a_1 \\ b_0 + 1.5c_0 + 1.6875d_0 & = & b_1 \\ 2c_0 + 4.5d_0 & = & 2c_1 \\ 2c_0 & = & 0 \\ 2c_1 + 4.5d_1 & = & 0 \end{array} \right.$$

La solución de este sistema es $a_0 = 0$, $b_0 = -0.521299$, $c_0 = 0$, $d_0 = 0.829607$, $a_1 = -0.0409838$, $b_1 = 0.878663$, $c_1 = 1.86662$, y $d_1 = -0.829607$. Es decir,

$$S(x) = \begin{cases} S_0(x) &= -0.521299x + 0.829607x^3 \text{ si } x \in [0, 0.75] \\ S_1(x) &= -0.0409838 + 0.878663(x - 0.75) + 1.86662(x - 0.75)^2 - 0.829607(x - 0.75)^3 \text{ si } x \in [0.75, 1.5]. \end{cases}$$

La representación gráfica para de S y f es



En las figuras (6.14) se muestra el trazador correspondiente a los nodos $x_0 = 0$, $x_1 = 0.5$, $x_2 = 1$, $x_3 = 1.5$ y $x_0 = 0$, $x_1 = 0.375$, $x_2 = 0.75$, $x_3 = 1.125$, $x_4 = 1.5$.

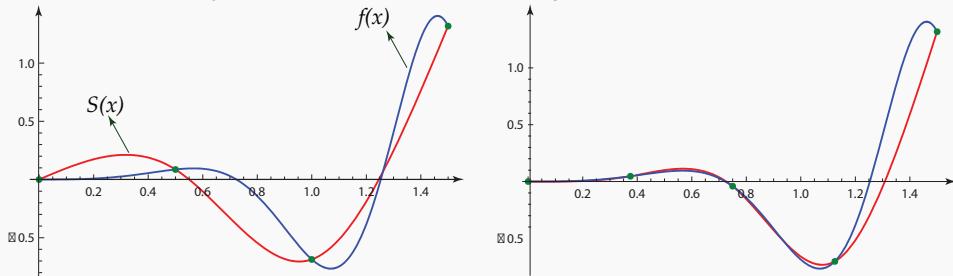
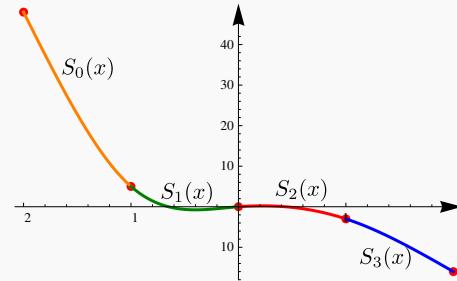


Figura 6.14: Trazador S y f con 3 y 4 puntos

Ejemplo 6.20

Determinar el trazador cúbico (frontera libre) para la siguiente tabla,

x_i	$y_i = x_i^4 - 4x_i^3$
$x_0 = -2$	48
$x_1 = -1$	5
$x_2 = 0$	0
$x_3 = 1$	-3
$x_4 = 2$	-16



Observe que la función $x^4 - 4x^3$ tiene un punto de inflexión en $x = 0$.

Solución: El trazador es,

$$S(x) = \begin{cases} S_0(x) = 9.85714(x+2)^3 - 52.8571(x+2) + 48 & \text{si } x \in [-2, -1], \\ S_1(x) = -11.2857(x+1)^3 + 29.5714(x+1)^2 - 23.2857(x+1) + 5 & \text{si } x \in [-1, 0], \\ S_2(x) = -0.714286x^3 - 4.28571x^2 + 2x & \text{si } x \in [0, 1], \\ S_3(x) = 2.14286(x-1)^3 - 6.42857(x-1)^2 - 8.71429(x-1) - 3 & \text{si } x \in [1, 2]. \end{cases}$$

■ **Pasos para obtener el trazador cúbico (frontera natural).** El proceso general sería como sigue. Sea $h_i = x_{i+1} - x_i$,

- De acuerdo al item (a) de la definición (6.15), $S_i(x_i) = y_i \implies a_i = y_i$.
- Haciendo algunas manipulaciones algebraicas en el sistema, se obtiene

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \quad \wedge \quad b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}). \quad (6.6)$$

La condición de frontera natural hace que $c_0 = c_n = 0$.

- Ahora todo depende del cálculo de los c_i 's. Éstos se calculan resolviendo el sistema $(n+1) \times (n+1)$

$$\left(\begin{array}{cccccc|c} 1 & 0 & 0 & \dots & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & \dots & 0 \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & \dots & 0 \\ \ddots & \ddots & \ddots & \ddots & & \\ 0 & \dots & h_{n-3} & 2(h_{n-3} + h_{n-2}) & h_{n-2} & \dots & 0 \\ 0 & 0 & 0 & & & \dots & 1 \end{array} \right) \cdot \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{pmatrix} = \begin{pmatrix} 0 \\ 3(f[x_2, x_1] - f[x_1, x_0]) \\ 3(f[x_3, x_2] - f[x_2, x_1]) \\ \vdots \\ 3(f[x_n, x_{n-1}] - f[x_{n-1}, x_{n-2}]) \\ 0 \end{pmatrix}.$$

Como antes, $f[x_i, x_j] = (y_i - y_j)/(x_i - x_j)$.



■ **Programa en Internet:** Trazadores cúbicos

Ejemplo 6.21

Determinar el trazador cúbico (frontera natural) para la siguiente tabla,

x_i	$y_i = \cos(3x_i^2) \ln(x_i^3 + 1)$
$x_0 = 0$	0
$x_1 = 0.5$	0.0861805
$x_2 = 1$	-0.686211
$x_3 = 1.5$	1.31799

Solución: El trazador es,

$$S(x) = \begin{cases} S_0(x) = a_0 + b_0(x - x_0) + c_0(x - x_0)^2 + d_0(x - x_0)^3 & \text{si } x \in [x_0, x_1], \\ S_1(x) = a_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 & \text{si } x \in [x_1, x_2]. \\ S_2(x) = a_2 + b_2(x - x_1) + c_2(x - x_1)^2 + d_2(x - x_1)^3 & \text{si } x \in [x_2, x_3]. \end{cases}$$

Hay que determinar los coeficientes de S_0 , S_1 y S_2 . Iniciamos calculando los c_i 's. Resolvemos el sistema 4×4 . Recordemos que $h_i = x_{i+1} - x_i$,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & 0 \\ 0 & h_1 & 2(h_1 + h_2) & h_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 3\left(\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}\right) \\ 3\left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}\right) \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 2 & 0.5 & 0 \\ 0 & 0.5 & 2 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ -5.15143 \\ 16.6596 \\ 0 \end{pmatrix}$$

Obtenemos $c_0 = 0$, $c_1 = -4.96871$, $c_2 = 9.57196$ y, por convenio, el comodín $c_3 = 0$.

Ahora podemos obtener el resto de coeficientes: $a_i = y_i$, los b_i 's y los d_i 's usando la ecuación (6.6).

$$b_0 = 1.00048, b_1 = -1.48387, b_2 = 0.8177508$$

$$d_0 = -3.31247, d_1 = 9.69378, d_2 = -6.38131.$$

Finalmente, el trazador cúbico es

$$\begin{cases} S_0(x) = -3.31247x^3 + 0.x^2 + 1.00048x & \text{si } x \in [0, 0.5], \\ S_1(x) = 9.69378(x - 0.5)^3 - 4.96871(x - 0.5)^2 - 1.48387(x - 0.5) + 0.0861805 & \text{si } x \in [0.5, 1], \\ S_2(x) = -6.38131(x - 1)^3 + 9.57196(x - 1)^2 + 0.8177508(x - 1) - 0.686211 & \text{si } x \in [1, 1.5]. \end{cases}$$

■ **Implementación en R.** En la base de **R** hay varias funciones para interpolar con trazadores cúbicos.

```
splinefun(x, y = NULL,
           method = c("fmm", "periodic", "natural", "monoH.FC", "hyman"),
           ties = mean)

spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x), xout, ties = mean)

splinefunH(x, y, m) # Interpolación Hermite. "m" es el vector de pendientes en cada punto
```

Por ejemplo, consideremos la función de Runge, $f(x) = \frac{1}{1+25x^2}$. Vamos a generar una tabla de puntos y usar trazadores cúbicos para generar un polinomio interpolante.

■ Código R 6.5:Ajuste de la función de Runge con Trazadores cúbicos

```
f = function(x) 1/(1+25*x^2)
n = 50
x = seq(-1,1, by=2/n)
y = f(x)
splinesRunge = splinefun(x,y, method="natural")
plot(x,y)
curve(splinesRunge(x), add=TRUE, col=2, n=1001)
```

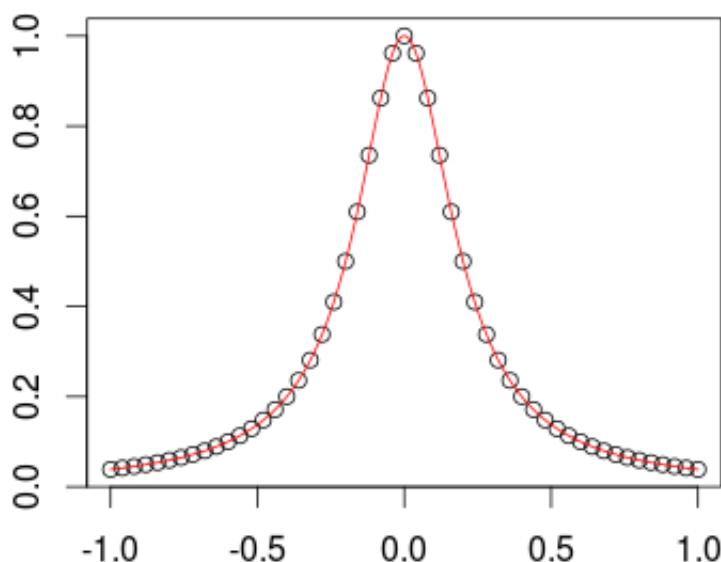


Figura 6.15: Trazador cúbico para la función de Runge

■ **Comparación.** Podemos usar las funciones `splinefun()` y `spline()` para hacer una comparación con el polinomio interpolante en el caso de muchos nodos.

```

require(PolynomF)
xi=c(0,.5,1,2,3,4)
yi=c(0,.93,1,1.1,1.15,1.2)
polyAjuste = poly.calc(xi,yi)
#polyAjuste
plot(xi,yi, pch = 19, cex=1.5, col= "red")
curve(polyAjuste,add=T,lty=2, lwd=4) # lwd = grosor, lty = tipo
# Usando trazadores cúbicos
splineAjuste=splinefun(xi,yi) # frontera natural
curve(splineAjuste,add=T,lty=1,lwd=4, col= "blue")
# splineAjuste.mono = splinefun(xi,yi,method="mono")#curva monótona
# curve(splineAjuste.mono,add=T,lty=1)
legend("bottomright",legend=c("Interpolante","Trazador"), lty=c(2:1),bty="n")

```

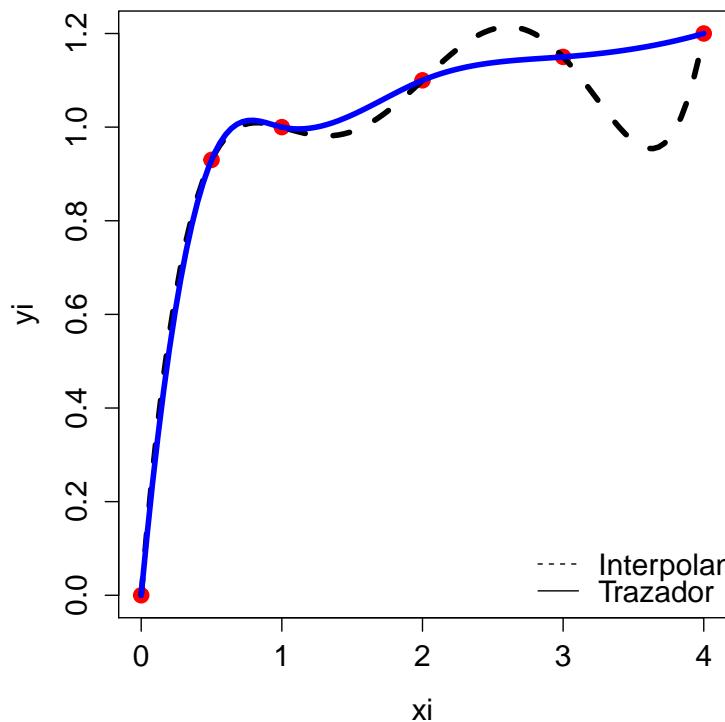


Figura 6.16: El polinomio interpolante es la gráfica en negro, en azul está el trazador cúbico.

8

Ejercicios

6.21 Calcule el trazador cúbico (natural) para el conjunto de datos $(0, 0), (1, 1), (2, 8)$.

6.22 Considere la tabla de datos,

$T(K)$	100	200	300	400	500	600
$B(cm^3/mol)$	-160	-35	-4.2	9.0	16.9	21.3

Figura 6.17: Segundos Coeficientes viriales $B(cm^3/mol)$ para el nitrógeno

donde T es la temperatura y B es el segundo coeficiente virial.

- Calcule el trazador cúbico (natural) para el conjunto de datos de la tabla.
- ¿Cuál es el segundo coeficiente virial (interpolado) a $450K$?
- Hacer la representación gráfica del trazador cúbico y del polinomio interpolante $P_5(x)$.

6.23 Considere la siguiente tabla de datos para el agua,

$T(C)$	50	60	65	75	80
$\rho(kg/m^3)$	988	985.7	980.5	974.8	971.6

donde T es temperatura y ρ es la densidad. Hacer la representación gráfica del trazador cúbico y del polinomio interpolante $P_4(x)$.



Revisado: Marzo, 2016

Versión actualizada de este libro:

<https://tecdigital.tec.ac.cr/revistamatematica/Libros/>

7 — Derivación e integración numérica

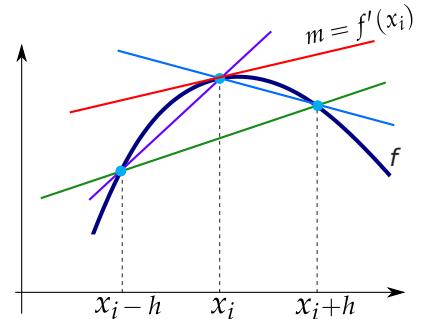
7.1 Derivación numérica

En esta sección solo vamos a considerar datos x_i igualmente espaciados. Sean $x_{i+1} = x_i + h$ y f derivable en un intervalo que contiene a x_i y a x_{i+1} . Entonces

$$f'(x_i) = \lim_{h \rightarrow 0} \frac{f(x_i + h) - f(x_i)}{h} \implies f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{h} \text{ si } h \text{ es suficientemente pequeño.}$$

Geométricamente podemos deducir tres aproximaciones:

- Diferencia finita hacia adelante: $f'(x_i) \approx \frac{f(x_i + h) - f(x_i)}{h}$
- Diferencia finita central: $f'(x_i) \approx \frac{f(x_i + h) - f(x_i - h)}{2h}$
- Diferencia finita hacia atrás: $f'(x_i) \approx \frac{f(x_i) - f(x_i - h)}{h}$



■ **Estimación del error.** Para establecer una estimación del error en la aproximación podemos deducir estas mismas fórmulas usando una expansión en serie de Taylor de f y usar el término de error de la expansión.

Sea f de clase $C^{n+1}[a, b]$. Supongamos que tenemos dos puntos x_i y x_{i+1} en este intervalo $[a, b]$. Sea $x_{i+1} = x_i + h$, es decir, $h = x_{i+1} - x_i$. La expansión en serie de Taylor de orden n para f es

$$f(x_i + h) = f(x_{i+1}) = f(x_i) + \frac{f'(x_i)}{1!}h + \frac{f''(x_i)}{2!}h^2 + \dots + \frac{f^{(n)}(x_i)}{n!}h^n + \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1} \text{ con } \xi \text{ entre } x_i \text{ y } x_{i+1}$$

También se puede reformular así:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) - \frac{h^5}{5!}f^{(5)}(x) + \dots$$

- Usando una expansión de orden dos, podemos despejar $f'(x_i)$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(\xi)}{2!}h^2 \text{ con } \xi \text{ entre } x_i \text{ y } x_{i+1}$$

y decimos que $f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{h}$ con error de truncación $-\frac{f''(\xi)}{2!}h^2 = O(h^2)$.

En general, una expresión del tipo $\frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$ decimos que es “de orden” $O(h^{n+1})$. En resumen, usando la expansión en serie de Taylor, podemos establecer algunas fórmulas de aproximación:

Fórmula de aproximación	Orden del error de truncación
Hacia adelante	

$$\textbf{(AD1)} \quad f'(x_j) = \frac{f(x_{j+1}) - f(x_j)}{h} \quad 0(h)$$

$$\textbf{(AD2)} \quad f'(x_j) = \frac{-f(x_{j+2}) + 4f(x_{j+1}) - 3f(x_j)}{2h} \quad 0(h^2)$$

$$\textbf{(AD3)} \quad f''(x_j) = \frac{f(x_{j+2}) - 2f(x_{j+1}) + f(x_j)}{h^2} \quad 0(h)$$

$$\textbf{(AD4)} \quad f''(x_j) = \frac{-f(x_{j+3}) + 4f(x_{j+2}) - 5f(x_{j+1}) + 2f(x_j)}{h^2} \quad 0(h^2)$$

Central

$$\textbf{(C1)} \quad f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} \quad O(h^2)$$

$$(C2) \quad f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12h} \quad O(h^4)$$

$$(C3) \quad f''(x_j) = \frac{f(x_{j+1}) - 2f(x_j) + f(x_{j-1})}{h^2} \quad O(h^2)$$

$$(C4) \quad f''(x_j) = \frac{-f(x_{j+2}) + 16f(x_{j+1}) - 30f(x_j) + 16f(x_{j-1}) - f(x_{j-2})}{12h^2} \quad O(h^4)$$

Hacia atrás

$$(AT1) \quad f'(x_j) = \frac{f(x_j) - f(x_{j-1})}{h} \quad O(h)$$

$$(AT2) \quad f'(x_j) = \frac{3f(x_j) - 4f(x_{j-1}) + f(x_{j-2})}{2h} \quad O(h^2)$$

$$(AT3) \quad f''(x_j) = \frac{f(x_j) - 2f(x_{j-1}) + f(x_{j-2})}{h^2} \quad O(h)$$

$$(AT4) \quad f''(x_j) = \frac{2f(x_j) - 5f(x_{j-1}) + 4f(x_{j-2}) - f(x_{j-3})}{h^2} \quad O(h^2)$$

Ejemplo 7.1

Con base en la siguiente tabla de datos (igualmente espaciados) para $f(x) = x \cos x$, vamos a calcular $f'(0.4)$ usando varios métodos.

x	0.1	0.2	0.3	0.4	0.5	0.6
$f(x)$	0.09950042	0.19601332	0.28660095	0.36842440	0.43879128	0.49520137

Tabla de comparación con $h = 0.1$

$h = 0.1$

Fórmula	Aproximación de $f'(0.4)$	Valor exacto $f'(0.4)$
(AD1)	0.7036688	0.7652937...
(AD2)	0.7734528	
(C1)	0.7609517	
(C2)	0.7652789	
(AT1)	0.8182345	
(AT2)	0.7744136	

Como $h = 0.1$ no es muy pequeño, las aproximaciones no son muy buenas excepto tal vez la que aproximación obtenida con la fórmula (C1).

■ **Derivación numérica con el paquete “numDeriv”** El paquete **numDeriv** de R usa algoritmos más exactos para hacer derivación numérica (primera y segunda derivada solamente), en particular de funciones escalares.

Para f' se usa la función **grad(fun, x, method= "...")**. Por defecto la función **grad()** usa método de “extrapolación de Richardson” que es muy preciso. También se puede elegir “**simple**” (derivación numérica hacia adelante con $h = 10^{-4}$) o “**complex**” que requiere que la función sea diferenciable de variable compleja y que x_i y $f(x_i)$ sean reales.

Para aproximar la segunda derivada se usa la función **hessian(func, x)**. Usa dos métodos, “Richardson” (por defecto) o “complex”.

■ Código R 7.1: Usando el paquete numDeriv

```
library(numDeriv) # también puede usar require(numDeriv)
f   = function(x) x*cos(x)
fp  = function(x) cos(x)-x*sin(x)
fpp = function(x) -sin(x)-sin(x)-x*cos(x)

options(digits=16)
# f^(1)(0.4)
grad(f,x=0.4)                      # Richardson
# [1] 0.7652936570778454
grad(f,x=0.4,method="simple") # diferencias hacia adelante, h = 10^-4
# [1] 0.765236289679283
fp(0.4)
# [1] 0.7652936570794249
# f^(2)(0.4)
hessian(f,x=0.4)
# [,1]
# [1,] -1.147261082222766
fpp(0.4)
# [1] -1.147261082218455
```

■ **Derivación numérica con el paquete “pracma”** . El paquete **pracma** también tiene varias funciones para hacer derivación numérica.

■ Código R 7.2: Usando el paquete pracma

```
f   = function(x) x*cos(x)
```

```

require(pracma)
options(digits=16)
fderiv(f, 0.4) # 1ra derivada
# [1] 0.765293657060566
fderiv(f, 0.4 , n = 2, h=1e-5) # 2da derivada con h = 10 ^-5
# [1] -1.147261730061188

```

Ejemplo 7.2 (Newton-Raphson con derivación numérica)

■ Código R 7.3: Método de Newton con derivación numérica

```

newtonDN = function(f, x0, tol, maxiter){
  # Derivada numérica con diferencia central
  fp = function(x) { h = 1e-15
    (f(x+h) - f(x-h)) / (2*h)
  }
  k = 0
  #Par imprimir estado
  cat("-----\n")
  cat(formatC( c("x_k"," f(x_k)","Error est."), width = -20, format = "f", flag = " "), "\n")
  cat("-----\n")

  repeat{
    correccion = f(x0)/fp(x0)
    x1 = x0 - correccion
    dx = corrección
    # Imprimir iteraciones
    cat(formatC( c(x1 ,f(x1), dx), digits=15, width = -15, format = "f", flag = " "), "\n")
    x0 = x1
    k = k+1
    # until
    if(dx <= tol || k > maxiter ) break;
  }
  cat("-----\n")
  if(k > maxiter){
    cat("Se alcanzó el máximo número de iteraciones.\n")
    cat("k = ", k, "Estado: x = ", x1, "Error estimado <= ", correccion)
  } else {
    cat("k = ", k, " x = ", x1, " f(x) = ", f(x1), " Error estimado <= ", correccion) 
  }
}

```

```

## --- Pruebas
f = function(x) x-cos(x)
options(digits = 15)
newtonDN(f, 0.5, 1e-10, 10)

# -----
#   x_k           f(x_k)      Error est.
# -----
# 0.751923064449049  0.021546382990783  0.251923064449049
# 0.738984893461253 -0.000167758744663  0.012938170987796
# 0.739085629223913  0.000000830126305  0.000100735762660
# 0.739085130749710 -0.000000004126207  0.000000498474203
# 0.739085133147489 -0.000000000113256  0.000000002397779
# 0.739085133215497  0.00000000000563   0.000000000068008
# -----
# k=6  x=0.739085133215497  f(x)=5.62883073484954e-13
# Error estimado <=-6.8007866666667e-11

```

Ejemplo 7.3

Considere el vector de datos x_i ,

```

xi = c(-0.41040018, -0.91061564, -0.61106896, 0.39736684, -0.37997637,
      0.34565436, -0.01906680, -0.28765977, -0.33169289, -0.99989810)

```

Ahora consideremos la función L definida como $L(\alpha) = \prod_{i=1}^n \frac{1 + \alpha \cdot x_i}{2}$ donde $n = \text{length}(xi)$. Podemos definir y hacer la representación gráfica de esta función con el código

```

xi = c(-0.41040018, -0.91061564, -0.61106896, 0.39736684, -0.37997637,
      0.34565436, -0.01906680, -0.28765977, -0.33169289, -0.99989810)
L = function(alpha) prod((1+alpha*xi)/2)
h = Vectorize(L)
curve(h, from = 1.2, to = 1.8, col = 2)
curve(0*x, from = 1.2, to = 1.8, col = 3, add=T)

```

Esta función tiene un cero en $[1.2, 1.8]$. Aunque podemos calcular la derivada de L , también podemos usar nuestra versión **newtonDN** para aproximar este cero,

```
newtonDN(L, 1.5, 1e-10, 15)

# -----
#   x_k           f(x_k)          Error est.
# -----
#   1.640904290093016 -0.000000062084156  0.140904290093016
#   1.636849885451859 -0.000000005228376  0.004054404641157
#   1.636507466091234 -0.000000000433406  0.000342419360625
#   1.636479070600085 -0.000000000035829  0.000028395491149
#   1.636476723126482 -0.000000000002962  0.000002347473603
#   1.636476529073748 -0.000000000000245  0.000000194052734
#   1.636476513032669 -0.000000000000020  0.000000016041078
#   1.636476511706660 -0.000000000000002  0.000000001326010
#   1.636476511597047 -0.000000000000000  0.000000000109612
#   1.636476511587987 -0.000000000000000  0.000000000009061
# -----
# k = 10 x = 1.63647651158799 f(x) = -1.14319400875298e-17
# Error estimado <= 9.0606666663841e-12
```

7.2 Integración numérica

La integral definida $\int_a^b f(x) dx$ no siempre se puede calcular usando el teorema fundamental del cálculo porque hay funciones que no tienen primitiva elemental, es decir la integral indefinida no se puede expresar en términos de funciones elementales. En estos casos, las integrales definen una nueva función.

Ejemplo 7.4

- a.) $\int e^{x^2} dx = \frac{\sqrt{\pi}}{2} \operatorname{Erfi}(x)$
- b.) $\int \frac{\sin x}{x} dx = \operatorname{SinIntegral}(x)$
- c.) $\int \frac{\cos x}{x} dx = \operatorname{CosIntegral}(x)$

d.) $\int \frac{e^x}{x} dx = \text{ExpIntegralEi}(x)$

e.) Integral de Fresnel $C(z) = \int_0^z \cos(\pi x^2/2) dx$

f.) etc.

Aquí solo consideramos métodos de integración aproximada de la forma

$$\int_a^b f(x) dx = \sum_{i=1}^n w_i f(x_i) \quad (7.1)$$

donde los *nodos* $x_0 < x_1 < x_2 < \dots < x_n$ están en $[a, b]$. A los w_i 's se les llama “pesos”.

7.3 Fórmulas de Newton-Cotes.

Las fórmulas de Newton-Cotes son fórmulas del tipo

$$\int_a^b w(x)f(x) dx = \sum_{i=0}^n w_i f(x_i) + E_n, \quad h = (b-a)/n, \quad x_i = a + i \cdot h.$$

Para determinar los pesos w_i se usa la fórmula de interpolación de Lagrange.

Sea $f \in C^{n+1}[a, b]$. Sea $P_n(x)$ el polinomio de grado $\leq n$ que interpola f en los $n+1$ puntos (distintos) x_0, x_1, \dots, x_n en el intervalo $[a, b]$. Para cada valor fijo $x \in [a, b]$ existe $\xi(x) \in]a, b[$ tal que

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n)$$

Entonces

$$\int_a^b f(x) dx = \int_a^b P_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n) dx \quad (7.2)$$

En particular, usando la forma de Lagrange del polinomio interpolante, $P_n(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + \dots + y_n L_{n,n}(x)$ con $L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i}$, tenemos

Teorema 7.1

Sea $f \in C^{n+1}[a, b]$. Sea $P_n(x)$ el polinomio de grado $\leq n$ que interpola f en los $n + 1$ puntos (distintos) x_0, x_1, \dots, x_n en el intervalo $[a, b]$. Existe $\eta \in]a, b[$ tal que

$$\int_a^b f(x) dx = \sum_{k=0}^n y_k \int_a^b \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} dx + \frac{f^{(n+1)}(\eta)}{(n+1)!} \int_a^b \prod_{i=0}^n (x - x_i) dx. \quad (7.3)$$

siempre y cuando $\prod_{i=0}^n (x - x_i)$ sea de un mismo signo en $[a, b]$.

También es de utilidad el siguiente teorema:

Teorema 7.2

Si $\int_a^b \prod_{i=0}^n (x - x_i) dx = 0$, $f \in C^{n+2}[a, b]$ y si $\prod_{i=0}^{n+1} (x - x_i)$ mantiene el mismo signo en $[a, b]$, entonces

$$\int_a^b f(x) dx = \sum_{k=0}^n y_k \int_a^b \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} dx + \frac{f^{(n+2)}(\eta)}{(n+2)!} \int_a^b \prod_{i=0}^{n+1} (x - x_i) dx, \quad \eta \in]a, b[\quad (7.4)$$

7.4 Regla del Trapecio.

En la regla del trapecio, para aproximar $\int_a^b f(x) dx$ dividimos el intervalo $[a, b]$ en n subintervalos: si $h = (b - a)/n$ y $x_i = a + i h$, en cada subintervalo $[x_i, x_{i+1}]$, cambiamos la función f por el polinomio interpolante de grado 1 (figura 7.1).



Figura 7.1: Regla del trapecio

Para aproximar cada integral $\int_{x_i}^{x_{i+1}} f(x) dx$ necesitamos el polinomio que interpola a f en $(x_i, f(x_i)), (x_{i+1}, f(x_{i+1}))$:

$$P(x) = f(x_i) \frac{(x - x_{i+1})}{h} + f(x_{i+1}) \frac{(x - x_i)}{h} + E \text{ con } E = (x - x_i)(x - x_{i+1}) \frac{f''(\xi(x))}{2} \text{ (si } f'' \text{ es continua en } [x_i, x_{i+1}]).$$

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x) dx &= \int_{x_i}^{x_{i+1}} f(x_i) \frac{(x - x_{i+1})}{h} + f(x_{i+1}) \frac{(x - x_i)}{h} + E dx \\ &= \frac{h}{2} [f(x_i) + f(x_{i+1})] + \int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1}) \frac{f''(\xi(x))}{2} dx\end{aligned}$$

Para calcular $\int_{x_i}^{x_{i+1}} E dx$ necesitamos recordar el teorema del valor medio para integrales: Si en $[x_i, x_{i+1}]$ G es continua y φ integrable y de un mismo signo, entonces existe $\eta_i \in]x_i, x_{i+1}[$ tal que $\int_{x_i}^{x_{i+1}} G(x)\varphi(x) dx = G(\eta_i) \int_{x_i}^{x_{i+1}} \varphi(x) dx$.

Ahora, poniendo $G(x) = f''(\xi(x))/2$ y $\varphi(x) = (x - x_i)(x - x_{i+1})$, obtenemos

$$\int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1}) \frac{f''(\xi_x)}{2} dx = -\frac{h^3}{12} f''(\eta_i), \quad \eta_i \in]x_i, x_{i+1}[.$$

Observe que $(x - x_i)(x - x_{i+1})$ tiene el mismo signo sobre $[x_i, x_{i+1}]$ (siempre es negativa) y que $\int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1}) dx = -h^3/6$.

Finalmente:

$$\int_{x_i}^{x_{i+1}} f(x) dx = \frac{h}{2} [f(x_i) + f(x_{i+1})] - \frac{h^3}{12} f''(\eta_i), \quad \eta_i \in [x_i, x_{i+1}]. \quad (7.5)$$

Si ponemos, para abreviar los cálculos, $G_i(x) = P_i(x) + (x - x_i)(x - x_{i+1})f''(\xi_i(x))/2$, con $P_i(x)$ el polinomio (lineal) interpolante en $[x_i, x_{i+1}]$, entonces

$$\begin{aligned}\int_a^b f(x) dx &= \int_{x_0}^{x_1} G_0(x) dx + \int_{x_1}^{x_2} G_1(x) dx + \dots + \int_{x_{n-1}}^{x_n} G_{n-1}(x) dx \\ &= \frac{h}{2} \left(f(x_0) + f(x_n) + 2 \sum_{i=1}^{n-1} f(x_i) \right) - \frac{h^3}{12} \sum_{k=0}^{n-1} f''(\eta_k)\end{aligned}$$

donde $h = \frac{b-a}{n}$, $\eta_i \in [x_i, x_{i+1}]$ y $x_i = a + i \cdot h$, $i = 0, 1, \dots, n$.

Podemos simplificar la fórmula observando que

$$-\frac{h^3}{12} \sum_{k=0}^{n-1} f''(\eta_k) = -\frac{h^2}{12} \cdot (b-a) \left[\frac{\sum_{k=0}^{n-1} f''(\eta_k)}{n} \right]$$

La expresión en paréntesis cuadrados es un *promedio* de los valores de f'' en $[a, b]$, por lo tanto este promedio está entre el máximo y el mínimo absoluto de f'' en $[a, b]$ (asumimos f'' continua). Finalmente, por el teorema del valor intermedio, existe $\xi \in]a, b[$ tal que $f''(\xi)$ es igual a este valor promedio, es decir

$$-\frac{h^3}{12} \sum_{k=0}^{n-1} f''(\eta_k) = -\frac{(b-a)h^2}{12} \cdot f''(\xi), \quad \xi \in]a, b[$$

(Regla compuesta del trapecio).

$$\int_a^b f(x) dx = \frac{h}{2} \left(f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right) - \frac{(b-a)h^2}{12} \cdot f''(\xi) \quad (7.6)$$

con $\xi \in]a, b[$, $h = \frac{b-a}{n}$ y $x_i = a + i \cdot h$, $i = 0, 1, 2, \dots, n$.

Ejemplo 7.5

Aunque sabemos que $\int_0^\pi \sin(x) dx = 2$, vamos a usar esta integral para ver como funciona la regla compuesta del trapecio. Aproximar $\int_0^\pi \sin(x) dx$ con tres subintervalos y estimar el error. Además determinar n tal que el error sea $|E| \leq 0.5 \times 10^{-8}$.

Solución: $n = 3$, $h = \frac{\pi - 0}{3} = \frac{\pi}{3}$, $x_0 = 0$, $x_1 = \frac{\pi}{3}$, $x_2 = \frac{2\pi}{3}$ y $x_3 = \pi$. Entonces,

$$\int_0^\pi \sin(x) dx \approx \frac{\pi/2}{3} [\sin(0) + \sin(\pi) + 2 \cdot (\sin(\pi) + \sin(2\pi/3))] = 1.813799364234\dots$$

El error estimado $|E|$, en valor absoluto, es $\leq \frac{\pi \cdot (\pi/3)^2}{12} M$ donde M es el máximo absoluto de $|f''(x)|$ en $[0, \pi]$. En este caso $M = 1$ y entonces $|E| \leq 0.287095\dots$

Para determinar n tal que $|E| \leq 0.5 \times 10^{-8}$. Procedemos así: Sabemos que el máximo absoluto de f'' en $[0, \pi]$ es $M = 1$, entonces

$$|E| \leq \frac{(b-a)h^2}{12} \cdot M = \pi \cdot \frac{(\pi/n)^2}{12} = \frac{\pi^3}{12n^2}$$

Como queremos $|E| \leq 0.5 \times 10^{-8}$, basta con que $\frac{\pi^3}{12n^2} \leq 0.5 \times 10^{-8}$. Despejando obtenemos,

$$n \geq \sqrt{\frac{\pi^3}{12 \cdot 0.5 \times 10^{-8}}} \approx 22732.603$$

Tomando $n = 22732$, obtenemos la aproximación $\int_0^\pi \sin(x) dx \approx 1.99999999681673$, que efectivamente tiene ocho decimales exactos.

Ejercicio 7.1 ■ Implementar la función **trapezio(fun, a,b, n)**.

Ejercicio 7.2 ■ Implementar la función **trapezio(xi,yi)** donde **(xi,yi)** son datos de una función f posiblemente no conocida. Observe que $y_i = f(x_i)$ y los x_i son datos igualmente espaciados con **a = min(xi)** y **b = max(xi)**

7.5 Regla del Simpson.

En vez de usar interpolación lineal, usamos interpolación cuadrática buscando una mejora en el cálculo. Por simplicidad, vamos a hacer el análisis en el intervalo $[x_0, x_2]$. Para construir la parábola que interpola f necesitamos los puntos x_0, x_1 y x_2 .

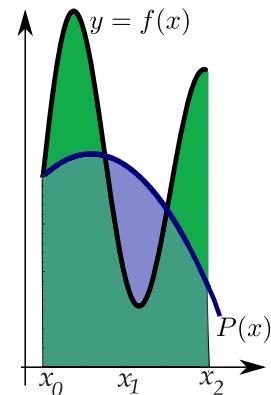


Figura 7.2

Sea $f^{(4)}$ continua en $[a, b]$. Interpolando f en $[a, b]$ con $x_0 = a$, $x_1 = (b+a)/2$ y $x_2 = b$ obtenemos el polinomio de Lagrange $P_2(x)$. Entonces,

$$f(x) = P_2(x) + f[x_0, x_1, x_2, x](x - x_0)(x - x_1)(x - x_2)$$

Luego,

$$\begin{aligned}\int_a^b f(x) dx &= \int_a^b P_2(x) + (x-x_0)(x-x_1)(x-x_2)f^{(4)}(\xi(x))/2 dx \\ &= \frac{h}{3} [f(a) + 4f(x_1) + f(b)] + \int_a^b (x-x_0)(x-x_1)(x-x_2)f^{(4)}(\xi(x))/2 dx\end{aligned}$$

En este caso, $\int_a^b (x-x_0)(x-x_1)(x-x_2) dx = 0$. Tomando $x_3 = x_1$, el polinomio $Q(x) = (x-a)\left(x-\frac{(a+b)}{2}\right)^2(x-b)$ es de un mismo signo sobre $[a, b]$. Aplicando el teorema (7.2) tenemos

$$\begin{aligned}\int_a^b f(x) dx &= \frac{h}{3} [f(a) + 4f(x_1) + f(b)] + \frac{f^{(4)}(\eta)}{4!} \int_a^b Q(x) dx, \quad \eta \in]a, b[\\ &= \frac{h}{3} [f(a) + 4f(x_1) + f(b)] - \frac{f^{(4)}(\eta)}{90} \left(\frac{b-a}{2}\right)^5, \quad \eta \in]a, b[\end{aligned}$$

pues $\int_a^b Q(x) dx = -\frac{4}{15} \left(\frac{b-a}{2}\right)^5$.

Para obtener la regla compuesta de Simpson necesitamos n **par** para poder escribir

$$\int_a^b f(x) dx = \int_{x_0}^{x_2} f(x) dx + \int_{x_2}^{x_4} f(x) dx + \dots + \int_{x_{n-2}}^{x_n} f(x) dx$$

Luego calculamos cada una de las $n/2$ integrales:

$$\int_{x_k}^{x_{k+2}} f(x) dx = \frac{h}{3} [f(x_k) + 4f(x_{k+1}) + f(x_{k+2})] - \frac{f^{(4)}(\eta_k)}{90} h^5$$

con $\eta_k \in]x_k, x_{k+2}[$ y $h = (x_{k+2} - x_k)/2$.

$$\begin{aligned}\int_a^b f(x) dx &= \int_{x_0}^{x_2} f(x) dx + \int_{x_2}^{x_4} f(x) dx + \dots + \int_{x_{n-2}}^{x_n} f(x) dx \\ &= \frac{h}{3} \left[f(a) + f(b) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) \right] - \sum_{k=1}^{n/2} \frac{f^{(4)}(\eta_k)}{90} h^5\end{aligned}$$

con $\eta_k \in]x_k, x_{k+2}[$ y $h = (b-a)/n$.

(Regla compuesta de Simpson). Si n es par,

$$\begin{aligned}\int_a^b f(x) dx &= \int_{x_0}^{x_2} f(x) dx + \int_{x_2}^{x_4} f(x) dx + \dots + \int_{x_{n-2}}^{x_n} f(x) dx \\ &= \frac{h}{3} \left[f(a) + f(b) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) \right] - \frac{1}{180} (b-a) h^4 f^{(4)}(\xi)\end{aligned}$$

con $\xi \in]a, b[$, $h = \frac{b-a}{n}$ y $x_i = a + i \cdot h$, $i = 0, 1, 2, \dots, n$.

La simplificación del término de error se hace como se hizo en la regla del Trapecio.

Aunque la regla de Simpson es muy popular en integración numérica (note que con la misma cantidad de evaluaciones obtenemos una aproximación con un error más pequeño) la regla del Trapecio es más eficiente en ciertas situaciones, como cuando trabajamos con polinomios trigonométricos.

Ejemplo 7.6

Aunque sabemos que $\int_0^\pi \sin(x) dx = 2$, vamos a usar esta integral para ver como funciona la regla de Simpson.

a.) Aproximar $\int_0^\pi \sin(x) dx$ con $n = 4$ y estimar el error.

b.) Estime n de tal manera que la regla de Simpson aproxime la integral con un error $|E| \leq 0.5 \times 10^{-8}$.

a.) **Solución:** Como $n = 4$, calculamos x_0, x_1, x_2, x_3 y x_4 .

$$n = 4 \implies h = \frac{\pi - 0}{4} = \frac{\pi}{4}, \quad x_0 = 0, \quad x_1 = \frac{\pi}{4}, \quad x_2 = \frac{\pi}{2}, \quad x_3 = \frac{3\pi}{4} \quad \text{y} \quad x_4 = \pi \quad \text{Entonces,}$$

$$\int_0^\pi \sin(x) dx \approx \frac{\pi/4}{3} [\sin(0) + \sin(\pi) + 4 \cdot (\sin(\pi/4) + \sin(3\pi/4)) + 2 \cdot \sin(\pi/2)] = 2.004559754984\dots$$

El error estimado $|E|$, en valor absoluto, es $\leq \frac{\pi \cdot (\pi/4)^4}{180} M$ donde M es el máximo absoluto de $|f^{(4)}(x)|$ en $[0, \pi]$. En este caso $M = 1$ y entonces $|E| \leq 0.00664105\dots$

b.) **Solución:**

Sabemos que el máximo absoluto de $f^{(4)}$ en $[0, \pi]$ es $M = 1$, entonces

$$|E| \leq \frac{(b-a)h^4}{180} \cdot M = \frac{\pi^5}{180n^4}$$

Como queremos $|E| \leq 0.5 \times 10^{-8}$, basta con que $\frac{\pi^5}{180n^4} \leq 0.5 \times 10^{-8}$. Despejando obtenemos,

$$n \geq \sqrt[4]{\frac{\pi^5}{180 \cdot 0.5 \times 10^{-8}}} \approx 135.79$$

Tomando $n = 136$, obtenemos la aproximación $\int_0^\pi \sin(x) dx \approx 2,0000000316395\dots$, que efectivamente tiene ocho decimales exactos.

Ejemplo 7.7

Determine un $n \in \mathbb{N}$ tal que al aproximar la integral

$$\int_{-1.5}^1 5 \cos(1-2x) - 2(x+1) \sin(1-2x) dx$$

con el método de Simpson, el error estimado sea menor o igual a $\delta = 0.5 \times 10^{-7}$. Para esto, use la fórmula de error de este método. Sugerencia: $f^{(4)}(x) = 16 \cos(1-2x) - 32(x+1) \sin(1-2x)$ y $f^{(5)}(x) = 64(x+1) \cos(1-2x)$.

Solución: Sea $M = \max|f^{(4)}(x)|$ en $[-1.5, 4]$. Entonces, como $b-a=2.5$ y $h=\frac{2.5}{n}$, el $n \in \mathbb{N}$ buscado debe ser par y cumplir

$$\frac{(2.5)\left(\frac{2.5}{n}\right)^4}{180} \cdot M \leq 0.5 \times 10^{-7}$$

Cálculo de M .

- Puntos críticos:

$$\begin{aligned} f^{(5)}(x) = 0 &\implies 64(x+1) \cos(1-2x) = 0 \implies \begin{cases} x = -1 \\ 1-2x = \frac{\pi}{2} \end{cases} \\ &\implies \begin{cases} x = -1 \in [-1.5, 1] \checkmark \\ x = \frac{2-\pi}{4} + 2k\pi, k \in \mathbb{Z} \end{cases} \end{aligned}$$

Las soluciones de la ecuación $\cos(1-2x) = 0$ son

k	...	-2	-1	0	1	2	...
$\frac{2-\pi}{4} + 2k\pi$...	-12.8518	-6.56858	-0.285398	5.99779	12.281	...

- Comparación: Los puntos críticos son $x = -1$ y $x = -0.285398\dots$

$$M = \max\{|f^{(4)}(-1.5)|, |f^{(4)}(1)|, |f^{(4)}(-1)|, |f^{(4)}(-0.2853981633974)|\} = 62.4989799215956$$

Por tanto, $\frac{(2.5)\left(\frac{2.5}{n}\right)^4}{180} \cdot 62.4989799215956 \leq 0.5 \times 10^{-7} \Rightarrow n \geq 161.374$

R/ $n = 162$

Simpson: Algoritmo e Implementación.

Notemos que no se requiere que n sea par, podríamos multiplicar por 2 y resolver el problema. Sin embargo vamos a suponer que n es par para tener control. Una manera directa de implementar la regla adaptativa de Simpson es alternar los x_i de subíndice par y los x_j de subíndice impar y multiplicarlos por 4 y 2 respectivamente. Esto se puede hacer en una sola línea.

Algoritmo 7.1: Regla adaptativa de Simpson

Datos: $f(x)$, a , b y $n \in \mathbb{N}$ par.

Salida: Aproximación de $\int_a^b f(x) dx$.

```

1 suma=0;
2 h=(b-a)/n;
3 for i = 0 to(n-1) do
4   suma=suma+f[a+i·h]+4·f[a+(i+1)·h]+f[a+(i+2)·h];
5   i = i + 2;
6 return suma·h/3.0

```

En R podemos hacer una implementación vectorizada,

■ Código R 7.4: Regla de Simpson

```

simpson = function(fun, a,b, n) {
  if (n%%2 != 0) stop("En la regla de Simpson, n es par!")
  h = (b-a)/n

```

```

i1 = seq(1, n-1, by = 2) # impares
i2 = seq(2, n-2, by = 2) # pares
h/3 * ( fun(a) + fun(b) + 4*sum( fun(a+i1*h) ) + 2*sum( fun(a+i2*h) ) )
}

##-----#
f = function(x) exp(-x) * cos(x)
simpson(f,0,pi,10)
# [1] 0.5214968
simpson(sin,0,pi,10)
# [1] 2.00011
simpson(sin,0,pi,100)
# [1] 2

```

7.6 Método de Romberg.

El método de Romberg usa la regla compuesta del trapecio para obtener aproximaciones preliminares y luego el proceso de extrapolación de Richardson para mejorar las aproximaciones.

Extrapolación de Richardson.

Supongamos que queremos estimar I y que podemos expresar I como

$$I = T(h) + a_2 h^2 + a_4 h^4 + a_6 h^6 + \dots \quad (7.7)$$

En este caso $T(h)$ es una aproximación de I y $a_4 h^4 + a_6 h^6 + \dots$ es el error de la estimación.

Supongamos además que T solo se puede evaluar para $h > 0$ (sino, el error sería nulo y no habría nada qué hacer) por lo que lo único que podemos hacer es tomar valores cada vez más pequeños de h .

Si $h \rightarrow 0$ entonces las potencias h^4, h^6, \dots se hacen pequeñas rápidamente por lo que, en la expresión del error $a_4 h^4 + a_6 h^6 + \dots$, el sumando que aporta la mayor parte del error es $a_4 h^4$ (si $a_4 \neq 0$). En un primer paso, el método de Romberg pretende mejorar la estimación de L eliminando este sumando. Para hacer esto procedemos así: En la ecuación (7.7) sustituimos h por $h/2$, luego podemos eliminar el sumando h^2 multiplicando esta última expresión por 4 y restando la expresión (7.7):

$$4I = 4T(h/2) + 4a_2 h^2/2^2 + 4a_4 h^4/2^4 + 4a_6 h^6/2^6 + \dots$$

$$-I = -T(h) - a_2 h^2 - a_4 h^4 - a_6 h^6 + \dots$$

Sumando y despejando obtenemos

$$I = \frac{4}{3} T(h/2) - \frac{1}{3} T(h) - a_4 h^4/2^2 - 5a_6 h^6/16 + \dots \quad (7.8)$$

¿Cuál es la ganancia? I ahora se aproxima con $\frac{4}{3}T(h/2) - \frac{1}{3}T(h)$ con un error más pequeño: $-a_4h^4/2^2 - 5a_6h^6/16 + \dots$.

Usando la notación O -grande diríamos que en (7.7) el error es de orden $O(h^2)$ (pues h^2 es la potencia dominante) mientras que en (7.8) expresión el error es de orden $O(h^4)$.

Ahora aplicamos el mismo procedimiento a $I = T_1(h) - b_4h^4 - b_6h^6 + \dots$, con $T_1(h) = 4/3T(h/2) - 1/3T(h)$, $b_4 = a_4/4$, $b_6 = a_6/2^3 \dots$

Para eliminar b_4h^4 cambiamos h por $h/2$ y multiplicamos por 16 y restamos la ecuación inicial

$$16I = 16T_1(h/2) - 16b_4h^4/2^4 - 16b_6h^6/64 + \dots$$

$$-I = -T_1(h) - b_4h^4 - b_6h^6 + \dots$$

Sumando y despejando obtenemos

$$I = \frac{16}{15}T_1(h/2) - \frac{1}{15}T_1(h) - b_6h^6/20 + \dots$$

Poniendo $T_2(h) = \frac{16}{15}T_1(h/2) - \frac{1}{15}T_1(h)$, entonces podemos decir que $T_2(h)$ aproxima I con un error de orden $O(h^6)$.

Siguiendo este procedimiento obtenemos $I = \frac{4^k}{4^k - 1}T_k(h/2) - \frac{1}{4^k - 1}T_k(h) + O(h^{2(k+1)})$ en el paso k -ésimo. Esto se puede simplificar y poner como

$$I = T_k(h/2) + \frac{T_k(h/2) - T_k(h)}{4^k - 1} + O(h^{2(k+1)})$$

7.6.1 Matriz de Romberg

El método de Romberg es una aplicación sistemática de esta idea de obtener una aproximación mejorada a partir de aproximaciones anteriores, iniciando con estimaciones de la regla del Trapecio para $h_k = \frac{b-a}{2^{k-1}}$, $k = 1, 2, \dots$

Sea $h_k = \frac{b-a}{2^{k-1}}$, $I = \int_a^b f(x) dx$ y $T(h) = \frac{h}{2} \left(f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right)$. Para aplicar el proceso de extrapolación de Richardson necesitamos

$$I = T(h) + a_2h^2 + a_4h^4 + a_6h^6 + \dots + a_{2m-2}h^{2m-2} + a_{2m}h^{2m}f^{(2m)}(\varepsilon) \quad (7.9)$$

Esto es cierto, pero para justificarlo necesitamos la fórmula de Euler-Maclaurin ([1]), así que aquí vamos a asumir este hecho.

El método de Romberg construye una matriz $R = (R_{i,j})$ en la que todas sus columnas convergen a I (las entradas son sumas de Riemann) pero la rapidez de convergencia crece de una columna a otra y esto se logra usando extrapolación de Richardson.

Trapezio	Extrapolación	Extrapolación
$R_{1,1}$		
$R_{2,1}$	$R_{2,2}$	
$R_{3,1}$	$R_{3,2}$	$R_{3,3}$
\vdots	\vdots	\vdots
$R_{n,1}$	$R_{n,2}$	$R_{n,3}$
\vdots	\vdots	\vdots
\downarrow	\downarrow	\downarrow
I	I	I

La primera columna de la matriz son los resultados de aplicar regla compuesta del trapecio: Se elige $h = b - a$ y se aplica regla del trapecio con $\text{h}_k = \frac{b-a}{2^{k-1}}$, $k = 1, 2, \dots$. La notación $R_{k,1}$ corresponde a la aproximación por Trapecios.

- $R_{1,1} = \frac{\text{h}_1}{2} [f(a) + f(b)] = \frac{b-a}{2} [f(a) + f(b)]$
- $R_{k,1} = \frac{1}{2} \left[R_{k-1,1} + \text{h}_{k-1} \sum_{i=1}^{2^{k-2}} f[a + (2i-1) \cdot \text{h}_k] \right], \quad k = 2, 3, \dots, n.$

Observe que $R_{k,1}$ es un fórmula recursiva para la regla compuesta del trapecio.

En particular,

$$\begin{aligned} R_{1,1} &= \frac{b-a}{2} [f(a) + f(b)] \\ R_{2,1} &= \frac{(b-a)}{4} \left[f(a) + f(b) + 2f\left(\frac{a+b}{2}\right) \right] \\ R_{3,1} &= \frac{1}{2} \left[R_{2,1} + \frac{a+b}{2} \left[f\left(\frac{3a+b}{4}\right) + f\left(\frac{a+3b}{4}\right) \right] \right] \end{aligned}$$

Luego, haciendo $h_{k+1} = h_k/2$ podemos obtener nuevas aproximaciones $R_{k,k}$ de manera recursiva (usando extrapolación), de la siguiente manera,

$$R_{k,j} = R_{k,j-1} + \frac{R_{k,j-1} - R_{k-1,j-1}}{4^{j-1} - 1}$$

En particular, si en la expansión (7.9) cada $a_i \neq 0$, entonces

$$R(n, m) = I + O\left(\frac{1}{2^{(n-1)(m-1)}}\right)$$

El cálculo se hace sencillo si formamos la matriz

$$R_{1,1}$$

$$R_{2,1} \quad R_{2,2}$$

$$R_{3,1} \quad R_{3,2} \quad R_{3,3}$$

$$R_{4,1} \quad R_{4,2} \quad R_{4,3} \quad R_{4,4}$$

$$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \ddots$$

$$\underline{R_{n,1} \quad R_{n,2} \quad R_{n,3} \quad R_{n,4} \quad \cdots \quad R_{n,n}}$$

Observe que el esquema de cálculo es similar al de diferencias divididas de Newton.

Ejemplo 7.8

Aunque sabemos que $\int_0^\pi \sin(x) dx = 2$, vamos a usar esta integral para ver como funciona la regla de Simpson. Aproximar $\int_0^\pi \sin(x) dx$ con $n = 4$ y $n = 6$ y estimar el error.

Solución: Calculamos la matriz de Romberg.

k			
1	0		
2	1.570796327	2.094395102	
3	1.896118898	2.004559755	1.998570732
4	1.974231602	2.00026917	1.999983131
			2.00000555

Tabla 7.3

Luego $\int_0^\pi \sin(x) dx \approx 2.000000016$. La estimación del error de truncación requiere comparar los elementos consecutivos de la última fila y escoger el menor

$$|R_{4,1} - R_{4,2}| = 0.0260376 \quad |R_{4,2} - R_{4,3}| = 0.000286039 \quad \text{y} \quad |R_{4,2} - R_{4,3}| = 0.000286039.$$

Así que la estimación del error en la truncación es de ± 0.000022419 .

Algoritmo e implementación.

Podemos usar $\text{Mín}_k \{R_{n,k} - R_{n,k-1}\}$ (el mínimo de las restas de cada dos columnas consecutivas en la fila n) como un estimado del error de truncación aunque frecuentemente éste resulta en una sobreestimación.

En la implementación del método de Romberg podríamos usar como criterio de parada: calcular la fila $n > 1$ hasta que $\text{Mín}_k \{R_{n,k} - R_{n,k-1}\} < \delta$ o $n \leq \text{numIter}$. Aquí numIter no debería ser más grande que, digamos 15. Además se debe indicar un número mínimo de iteraciones, digamos $n = 3$. Se trata de una heurística para evitar la finalización prematura cuando el integrando oscila mucho.

Ejercicio 7.3 ■ Implementar la función `romberg(fun,a,b,n)`

7.7 Cuadratura Gaussiana.

En la cuadratura Gaussiana (método de Gauss para aproximar una integral), en vez de usar una partición igualmente espaciada del intervalo $[a, b]$ para aproximar la integral con $n+1$ puntos, se escogen los “mejores” $x_0, x_1, \dots, x_n \in [a, b]$, de tal manera que la aproximación sea exacta al menos, para polinomios de grado menor o igual a $2n+1$ (recordemos que Trapecio es exacto para polinomios de grado 1 y Simpson para polinomios de grado 3).

En la figura 7.3, el área de la región que cubre el trapecio relleno es exactamente el área de la región entre la parábola y el eje X . La aproximación que da la regla del Trapecio (trapecio punteado) no es exacta en este caso.

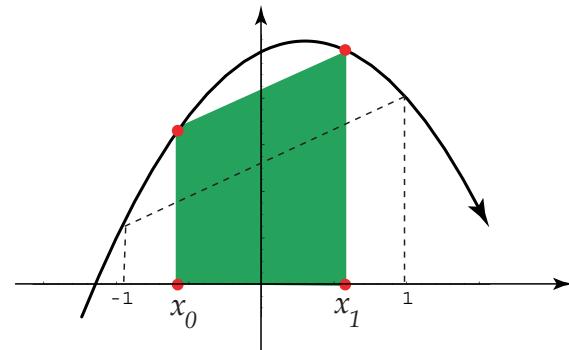


Figura 7.3: Cuadratura Gaussiana.

En general, se trata de usar el método de *coeficientes indeterminados*: determinar c_0, c_1, \dots, c_n y $x_0, x_1, \dots, x_n \in [-1, 1]$ de tal manera que las integrales

$$\int_{-1}^1 P_{2n+1}(x) dx = c_0 f(x_0) + c_1 f(x_1) + \dots + c_n f(x_n)$$

son exactas para cada $P_{2n+1}(x)$, un polinomio de grado $2n+1$, $n = 0, 1, 2, \dots$

Debemos resolver el sistema (no lineal) con $n+1 + n+1 = 2n+2$ incógnitas,

$$\left\{ \begin{array}{lcl} c_0 f(x_0) + c_1 f(x_1) + \dots + c_n f(x_n) & = & \int_{-1}^1 1 \, dx = 2, & f(x) = 1 \\ c_0 f(x_0) + c_1 f(x_1) + \dots + c_n f(x_n) & = & \int_{-1}^1 x \, dx = 0, & f(x) = x \\ \vdots & & \vdots & \\ c_0 f(x_0) + c_1 f(x_1) + \dots + c_n f(x_n) & = & \int_{-1}^1 x^{2n} \, dx = \frac{2}{2n+1}, & f(x) = x^{2n} \\ c_0 f(x_0) + c_1 f(x_1) + \dots + c_n f(x_n) & = & \int_{-1}^1 x^{2n+1} \, dx = 0, & f(x) = x^{2n+1} \end{array} \right.$$

En la tabla que sigue, aparecen la solución aproximada, hasta $n = 5$.

n	c_i	x_i
1	$c_0 = 1$	$x_0 = -0.5773503$
	$c_1 = 1$	$x_1 = -0.5773503$
2	$c_0 = 0.5555555556$	$x_0 = -0.7745966692$
	$c_1 = 0.8888888889$	$x_1 = 0$
	$c_2 = 0.5555555556$	$x_2 = 0.7745966692$
3	$c_0 = 0.3478548451$	$x_0 = -0.8611363116$
	$c_1 = 0.6521451549$	$x_1 = -0.3399810436$
	$c_2 = 0.6521451549$	$x_2 = 0.3399810436$
	$c_3 = 0.3478548451$	$x_3 = 0.8611363116$
4	$c_0 = 0.2369268850$	$x_0 = -0.9061798459$
	$c_1 = 0.4786286705$	$x_1 = -0.5384693101$
	$c_2 = 0.5688888889$	$x_2 = 0$
	$c_3 = 0.4786286705$	$x_3 = 0.5384693101$
	$c_4 = 0.2369268850$	$x_4 = 0.9061798459$
5	$c_0 = 0.1713245$	$x_0 = -0.932469514$
	$c_1 = 0.3607616$	$x_1 = -0.661209386$
	$c_2 = 0.4679139$	$x_2 = -0.238619186$
	$c_3 = 0.4679139$	$x_3 = 0.238619186$
	$c_4 = 0.3607616$	$x_4 = 0.661209386$
	$c_5 = 0.1713245$	$x_5 = 0.932469514$

Tabla 7.4

Una tabla más extensa se puede encontrar en [Proyecto Euclides](#).

Se puede probar que x_0, x_1, \dots, x_n son las raíces de los polinomios de Legendre,

$$P_{n+1}(x) = \frac{(n+1)!}{(2n+2)!} G_n^{(n+1)}(x) \quad n = 1, 2, \dots$$

con $G_n^{(n+1)}(x)$ la derivada $n+1$ de $G_n(x) = (x^2 - 1)^{n+1}$.

Si tenemos las raíces (que son todas reales), los c_i 's se podrían calcular con la fórmula

$$c_i = \int_{-1}^1 \prod_{j=0, j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)} dx$$

(Cuadratura Gaussiana).

- Para calcular en un intervalo $[a, b]$ usando cuadratura Gausiana hacemos el cambio de variable $x = \frac{a + b + (b - a)u}{2}$ y teniendo en cuenta que $dx = \frac{b - a}{2} du$, obtenemos

$$\int_a^b f(x) dx = \int_{-1}^1 \frac{b - a}{2} f\left(\frac{a + b + (b - a)u}{2}\right) du = c_0 g(x_0) + c_1 g(x_1) + \dots + c_n g(x_n) + E_n$$

donde, por supuesto, $g(u) = \frac{b - a}{2} f\left(\frac{a + b + (b - a)u}{2}\right)$.

- Si $g \in C^{2n}[-1, 1]$, el error en la fórmula de cuadratura Gaussiana es ([13]),

$$E_n = \frac{2^{2n+1}[(n)!]^4}{(2n+1)[(2n)!]^3} g^{(2n)}(\xi) \text{ con } \xi \in]-1, 1[.$$

Ejemplo 7.9

Aproximar $\int_0^\pi \sin(x) dx$ con $n = 3$ y estimar el error.

Solución:

- En este caso, el cambio de variable es $x = \frac{0 + \pi + (\pi - 0)u}{2} = \frac{\pi + \pi u}{2}$ y $dx = \frac{\pi}{2} du$.
- $g(u) = \frac{\pi}{2} \sin\left(\frac{\pi + \pi u}{2}\right) = \frac{\pi}{2} \cos\left(\frac{\pi u}{2}\right)$.

$$\begin{aligned} \int_0^\pi \sin(x) dx &= \int_{-1}^1 \frac{\pi}{2} \cos(\pi u/2) du \\ &\approx c_0 g(x_0) + c_1 g(x_1) + c_2 g(x_2) + c_3 g(x_3) \\ &= \pi/2 \left[c_0 \cos\left(\frac{\pi x_0}{2}\right) + c_1 \cos\left(\frac{\pi x_1}{2}\right) + c_2 \cos\left(\frac{\pi x_2}{2}\right) + c_3 \cos\left(\frac{\pi x_3}{2}\right) \right] \\ &\quad \text{y usando la tabla de valores,} \\ &= 1.9999842284987... \end{aligned}$$

La estimación del error es,

$$\begin{aligned}|E_3| &= \frac{2^7 \cdot (3!)^4}{7 \cdot (6!)^3} \cdot |g^{(6)}(\xi)| \text{ con } \xi \in]-1, 1[. \\ &\leq \frac{2^7 \cdot (3!)^4}{7 \cdot (6!)^3} \cdot \frac{\pi^7}{128} \approx 0.00149816\end{aligned}$$

pues $|g^{(6)}(u)| = \frac{\pi^7 \cos\left(\frac{\pi u}{2}\right)}{128} \leq \frac{\pi^7}{128}$ en $] -1, 1 [.$

7.8 Integrales Impropias.

Las integrales impropias (convergentes) $\int_a^\infty f(x) dx$, ($a > 0$) y $\int_{-\infty}^b f(x) dx$, ($b < 0$); se pueden calcular usando un cambio de variable.

Si $a > 0$ y $b = \infty$ o si $b < 0$ y $a = -\infty$ entonces,

$$\int_a^b f(x) dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt$$

Como $1/a$ o $1/b$ es cero, $f\left(\frac{1}{t}\right)$ se indefine. Así que no podemos considerar métodos de integración que evalúen los extremos (como Simpson o Trapecio) sino más bien, de acuerdo a lo que tenemos hasta aquí, cuadratura Gaussiana.

7.9 Integración con R

La función **integrate()**, en la base de **R**, calcula numéricamente la integral definida de una función y además hace una estimación del error. Si la función maneja vectores, se debe vectorizar con **Vectorize()** antes de usar **integrate()**.

■ Código R 7.5: Usando **integrate()**

```
f = function(x) exp(-x) * cos(x)
integrate(f, 0, pi)
# 0.521607 with absolute error < 7.6e-15

# Vectorize() si f opera con vectores
f1 = function(x) max(0, x)
h = Vectorize(f1)
integrate(h, -1, 1)
```

```
# 0.5 with absolute error < 5.6e-15

fgauss = function(t) exp(-t^2/2)
integrate(fgauss, -Inf, Inf)
# 2.506628 with absolute error < 0.00023
```

Para extraer solo el valor de la integral (sin el error estimado) usamos

```
integrate(fgauss, -Inf, Inf)$value
# [1] 2.506628
```

Ejemplo 7.10 (Ceros de una función definida por una integral)

La función $S(x) = \int_0^x t - \operatorname{sen}(t^2) - 1 dt$ no tiene primitiva elemental. Esta función tiene un cero cerca de $x = 2.5$. Podemos usar la función **newtonRaphson()** del paquete **pracma** para aproximar este cero. Esta función **newtonRaphson()** hace derivación numérica en el cálculo.

■ Código R 7.6: Cero de una función definida por una integral

```
st = function(t) t-sin(t^2)-1
S = function(x) integrate(st, 0, x)$value
# Graficar S(x)-----
h = Vectorize(S)
curve(h, 0,3)
#-----
require(pracma)
newtonRaphson(S, 2.5)

# $root
# [1] 2.388456

# $f.root
# [1] 6.297833e-16

# $niter
# [1] 5

# $estim.prec
# [1] 3.083087e-16
```

En el paquete **pracma** hay varias funciones para integrar, usando versiones mejoradas de los métodos que hemos visto más arriba. En particular hay métodos de integración para funciones discretas.

9

Ejercicios

7.1 Considere la integral $I = \int_0^1 e^{-x} dx$.

- Aproximar la integral con la regla del Trapecio para $n = 4$ y estime el error de la aproximación.
- Estime n de tal manera que, usando la regla del Trapecio, el error estimado es $\leq 0.5 \times 10^{-10}$. Use su implementación en Excel para calcular la aproximación correspondiente (de I).
- Aproximar la integral con la regla del Simpson para $n = 4$ y estime el error de la aproximación.
- Estime n de tal manera que, usando la regla del Simpson, el error estimado sea $\leq 0.5 \times 10^{-10}$. Use su implementación en Excel para calcular la aproximación correspondiente (de I).
- Aproximar la integral con el método de Romberg para $n = 4$ y estime el error de la aproximación.
- Use su implementación del método de Romberg para encontrar experimentalmente, usando la estimación del error, el n adecuado para el que el error estimado sea $\leq 0.5 \times 10^{-10}$.
- Aproximar la integral con cuadratura Gaussiana para $n = 4$ y estime el error de la aproximación.
- Estime n de tal manera que, usando cuadratura Gaussiana, el error estimado sea $\leq 0.5 \times 10^{-10}$.

Ayuda: aquí no se trata de hacer un despeje de n (por la presencia de factoriales) sino, más bien, ensayar (tanteo) con valores de n (en la fórmula del error) hasta lograr el objetivo. Observe que $g(u) = \frac{1}{2} e^{-(u+1)/2}$. Las derivadas de g tienen un patrón:

$$\begin{aligned}|g'(u)| &= \frac{1}{4} e^{-(u+1)/2} \\|g''(u)| &= \frac{1}{8} e^{-(u+1)/2} \\|g^{(3)}(u)| &= \frac{1}{16} e^{-(u+1)/2} \\|g^{(4)}(u)| &= \frac{1}{32} e^{-(u+1)/2} \\&\dots\end{aligned}$$

7.2 Aproximar $\int_0^1 \frac{\cos x}{\sqrt{x}} dx$ con el método de Simpson con $n = 4$.

7.3 Considere las integrales de Fresnel, $S(x) = \int_0^x \sin(t^2) dt$ y $C(x) = \int_0^x \cos(t^2) dt$. Se sabe que existe un valor $\xi \in [1, 2]$ tal que $S(\xi) = C(\xi)$. Aproxime este valor usando el método de la secante.

7.4 La “función error” se define como $\text{Erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. Aproximar $\text{Erf}(1.5)$ usando los cuatro métodos de integración hasta que la diferencia en cada resultado sea $\leq 0.5 \times 10^{-5}$.

7.5 De una función f , conocemos la siguiente información

x	$f(x)$
0	3.592
0.2	3.110
0.4	3.017
0.6	2.865
0.8	2.658

Tabla 7.5

a.) Aproximar $\int_0^{0.8} f(x) dx$ usando regla del Trapecio.

b.) Aproximar $\int_0^{0.8} f(x) dx$ usando regla del Simpson.

c.) Aproximar $\int_0^{0.8} f(x) dx$ usando Romberg (interpolar con polinomios de grado 2).

7.6 Aproxime $\int_1^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$ con $n = 6$.

7.7 Aproxime $\int_1^5 \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$ con $n = 6$.

7.8 La función de Bessel de orden cero se define como

$$J_0(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \operatorname{sen} t) dt$$

Derivando bajo el signo integral obtenemos $\frac{d}{dx} J_0(x) = \frac{1}{\pi} \int_0^{\pi} \frac{d}{dx} [\cos(x \operatorname{sen} t)] dt$

$$\begin{aligned}
 J_0'(x) &= -\frac{1}{\pi} \int_0^\pi \sin t \sin(x \sin t) dt \\
 J_0''(x) &= -\frac{1}{\pi} \int_0^\pi (\sin t)^2 \cos(x \sin t) dt \\
 &\vdots \quad \vdots \\
 |J_0^{(n)}(x)| &= \begin{cases} \frac{1}{\pi} \int_0^\pi |(\sin t)^n \cos(x \sin t)| dt & \text{si } n \text{ es par} \\ \frac{1}{\pi} \int_0^\pi |(\sin t)^n \sin(x \sin t)| dt & \text{si } n \text{ es impar} \end{cases}
 \end{aligned}$$

a.) Muestre que $|J_0^{(n)}(x)| \leq \frac{1}{\pi} \int_0^\pi 1 dt = 1$, $n = 0, 1, 2, \dots$. Sugerencia: Si f es continua en $[a, b]$ entonces

$$\min f \leq \frac{1}{b-a} \int_a^b f(x) dx \leq \max f$$

b.) Dado $\delta > 0$, determine n de tal manera que si aproximamos $J_0(x)$ con la regla compuesta de Simpson, el error sea $\leq \delta$.

- c.) Implemente en **R**, una función **J0Simpson(x, delta)** para aproximar $J_0(x)$, usando la regla de Simpson, con un error estimado $\leq \delta$.
- d.) Realice la representación gráfica de $J_0(x)$ con $x \in [-5, 5]$.
- e.) La función $J_0(x)$ tiene un cero x^* en $[2, 3]$. Implemente una versión del método de bisección y una versión del método de Newton que operen con la función **J0Simpson(x, 0.5*1e-5)** y aproxime en cada caso x^* con un error estimado $\leq 0.5 \times 10^{-8}$.

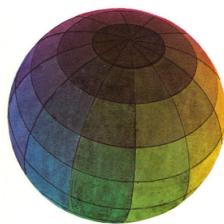
7.9 Considere la integral $I = \int_0^1 e^{-x^2} dx$.

- a.) Aproximar la integral con la regla compuesta de Simpson con $n = 4$
- b.) Estime el error en la aproximación anterior.
- c.) Estime n de tal manera que, usando la regla comuesta de Simpson, el error estimado de la aproximación sea $\leq 0.5 \times 10^{-10}$.

7.10 De una función f , conocemos la siguiente información

x	$f(x)$
0	3.5
0.2	3.1
0.4	3
0.6	2.8
0.8	2.6

Aproximar $\int_0^{0.8} f(x) dx$ usando regla compuesta de Simpson con $n = 4$.



Revisado: Marzo, 2016

Versión actualizada de este libro:

<https://tecdigital.tec.ac.cr/revistamatematica/Libros/>



8 — Ecuaciones Diferenciales Ordinarias

Consideremos el problema de valor inicial

$$\frac{dy}{dt} = f(t, y(t)), \quad a \leq t \leq b, \quad y(a) = y_0 \quad (8.1)$$

Buscamos una función $y(t) \in C^1[a, b]$ que satisfaga la identidad (8.1). Se asume que $f(t, y)$ está definida para $t \in [a, b]$ y $y \in \mathbb{R}^m$. Por supuesto, en este texto solo estudiamos el caso $m = 1$.

Existencia y unicidad. En teoría de ecuaciones diferenciales se establece el siguiente teorema,

Teorema 8.1

Si $f(t, y)$ es continua en $t \in [a, b]$ y respecto a y satisface la condición de Lipschitz

$$||f(t, y) - f(t, y^*)|| \leq L ||y - y^*||, \quad t \in [a, b], \quad y, y^* \in \mathbb{R},$$

entonces el problema de valor inicial (8.1) tiene una única solución $y(t)$, $a \leq t \leq b$, para cualquier $y_0 \in \mathbb{R}$.

Supongamos que tenemos el problema de valor inicial

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = y_0 \quad (8.2)$$

Si tenemos una aproximación (t_i, y_i) de $(t_i, y(t_i))$, el paso siguiente en un método de un solo paso es

$$y_{i+1} = y_i + h \cdot \Phi(t_i, y_i; h), \quad h > 0.$$

Solución numérica. Desde el punto de vista numérico lo que nos interesa encontrar aproximaciones $y_a(t_i)$ a los valores exactos $y(t_i)$. En este capítulo, los $t_i \in [a, b]$ los tomaremos igualmente espaciados, es decir, si $h = (b - a)/n$, $t_i = a + i \cdot h$, $i = 0, 1, \dots, n$.

Si uno lo prefiere, puede construir una tabla de aproximaciones $\{(t_i, y_a(t_i)), i = 0, 1, \dots, n\}$ y por interpolación, construir una solución aproximada $y_a(t)$, $t \in [a, b]$.

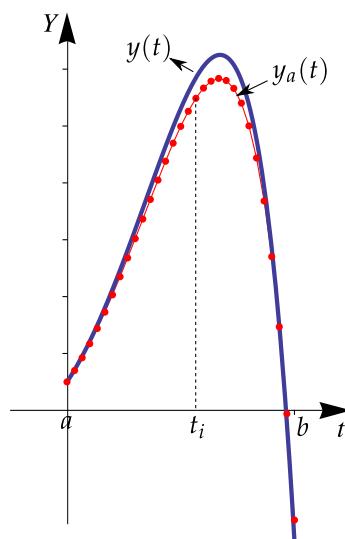


Figura 8.1: Solución numérica de un problema de valor inicial

La función Φ se puede ver como el incremento aproximado en cada paso y define cada método de un solo paso.

Orden del método. Para definir el orden del método necesitamos definir el *error de truncación*. Sea $u(t)$ la solución del problema 8.2 pero pasando por el punto (genérico) (x, y) , es decir, $u(t)$ es la solución del problema local

$$\frac{du}{dx} = f(t, u), \quad x \leq t \leq x + h, \quad u(x) = y \quad (8.3)$$

A $u(t)$ se le llama *solución de referencia*. Si $y^* = y + h\Phi(x, y; h)$, y^* aproxima $u(x + h)$ con un *error de truncación* $T(x, y; h) = \frac{1}{h}(y^* - u(x + h))$.

El método Φ se dice de orden p si $\|T(x, y; h)\| \leq Ch^p$ uniformemente sobre $[a, b]$ donde la constante C no depende de x, y o h . Esta propiedad es usual escribirla como

$$T(x, y; h) = O(h^p), \quad h \rightarrow 0,$$

es decir, entre más grande p , más exacto es el método.

A continuación, vamos a ver algunos métodos de un solo paso.

8.1 Método de Euler

Euler propuso este método en 1768. Consiste en seguir la tangente en cada punto (t_i, y_i) . Hacemos una partición del intervalo $[a, b]$ en n subintervalos $[t_i, t_{i+1}]$, cada uno de longitud $h = (b - a)/n$. Luego, $t_{i+1} = a + i \cdot h = t_i + h$.

Iniciando con (t_0, y_0) , se calcula la ecuación de la tangente en (t_i, y_i) : $y_T(t) = f(t_i, y_i)(t - t_i) + y_i$ y se evalúa en $t = t_{i+1} = t_i + h$, es decir,

$$y(t_{i+1}) \approx y_{i+1} = y_i + h f(t_i, y_i), \quad i = 0, 1, \dots, n$$

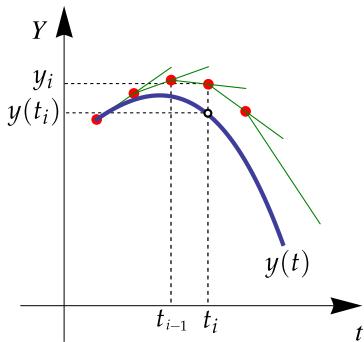


Figura 8.2: $y(t_i) \approx y_i = y_{i-1} + h f(t_{i-1}, y_i)$.

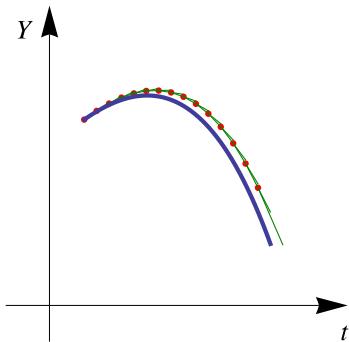


Figura 8.3: Tangentes en (t_i, y_i) , $i = 0, 1, \dots, 15$.

El método de Euler es de orden $p = 1$.

Ejemplo 8.1

Consideremos el problema de valor inicial $\frac{dy}{dt} = 0.7y - t^2 + 1$, $t \in [1, 2]$, $y(1) = 1$. Aquí $a = 1$, $b = 2$. Si $n = 10$ entonces $h = 0.1$ y $t_i = a + h i = 1 + 0.1 i$

$$\begin{cases} y_0 &= 1 \\ y_{i+1} &= y_i + h(0.7 * y_i - t_i^2 + 1) = y_i + 0.1(0.7y_i - (1 + 0.1i)^2 + 1), \quad i = 1, \dots, n \end{cases}$$

i	t_i	y_i
0	1	1
1	1.1	1.07
2	1.2	1.1239
3	1.3	1.15857
4	1.4	1.17067
5	1.5	1.15662
6	1.6	1.11258
7	1.7	1.03446
8	1.8	0.917877
9	1.9	0.758128
10	2.	0.550197

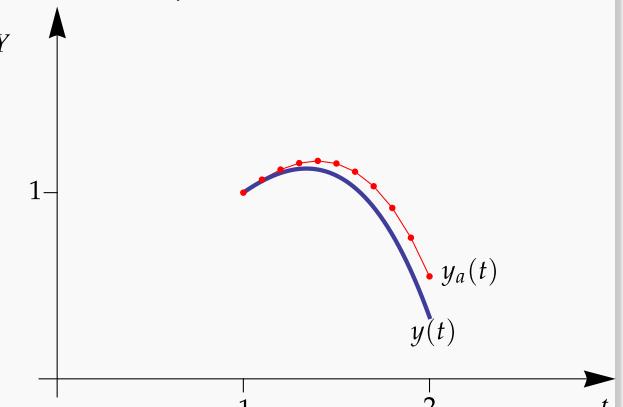


Tabla 8.1: $y(t_i) \approx y_i = y_{i-1} + h f(t_{i-1}, y_i)$, $i = 1, 2, \dots, 10$.

8.2 Algoritmo e implementación

En el algoritmo se usan los datos t_0, y_0, h y n . Se imprimen n datos (t_i, y_i) y el gráfico.

Algoritmo 8.1: Método de Euler

Datos: $f(t, y), a, b, y_0, n$
Salida: Imprime las aproximaciones $(t_i, y_i), i = 0, 1, \dots, n$

```

1   $h = (b - a) / n;$ 
2   $t_0 = a;$ 
3  for  $i = 1$  ton do
4       $y_1 = y_0 + h \cdot f(t_0, y_0);$ 
5       $t_0 = a + i \cdot h;$ 
6       $y_0 = y_1;$ 
7      print(( $t_0, y_0$ ));

```

■ Código R 8.1: Método de Euler

```

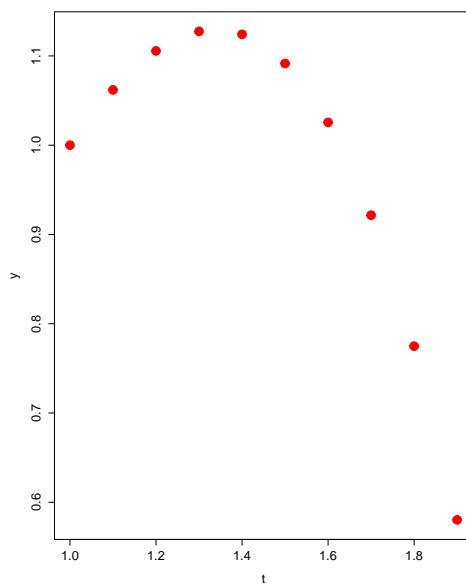
euler1 = function(f, t0, y0, h, n) {
  #Datos igualmente espaciados iniciando en x0 = a, paso h. "n" datos
  t = seq(t0, t0 + (n-1)*h, by = h) # n datos
  y = rep(NA, times=n) # n datos
  y[1]=y0
  for(i in 2:n ) y[i]= y[i-1]+h*f(t[i-1], y[i-1])
  print(cbind(t,y)) # print
  plot(t,y, pch=19, col="red") # gráfica
}

```

```

# --- Pruebas
f = function(t,y) -1.68*10^(-9)*y^4+2.6880
# t0 = 20; y0 = 180, paso h=10, n = 10 puntos (ti,yi)
)
euler1(f, 20, 180, 10, 8)
#      t      y
# [1,] 20 180.0000
# [2,] 30 189.2440
# [3,] 40 194.5765
# [4,] 50 197.3757
# [5,] 60 198.7590
# [6,] 70 199.4200
# [7,] 80 199.7304
# [8,] 90 199.8751

```



8.3 Métodos de Taylor de orden superior.

El método de Euler opera con un polinomio de Taylor de orden uno (rectas). Es natural, como propuso Euler, usar más términos e la expansión de Taylor (si f es suficientemente derivable). Usando una expansión de orden m nos lleva a un método de orden $O(h^m)$. El costo de calcular las derivadas en estos tiempos se le delega a los computadores, lo que hace que el método (todavía de un solo paso), sea una opción viable.

En este método, calculamos el polinomio de Taylor alrededor de $t = t_i$ (en potencias de $t - t_i$) y evaluamos este polinomio en $t_{i+1} = t_i + h$. Nos queda un polinomio en potencias de h ,

$$y(t_{i+1}) = y(t_i + h) = y(t_i) + h y'(t_i) + \frac{h^2}{2} y''(t_i) + \dots + \frac{h^m}{m!} y^{(m)}(t_i) + \frac{h^{m+1}}{(m+1)!} y^{(m+1)}(\xi_i), \quad \text{con } \xi_i \in]t_i, t_i + h[.$$

Como $y'(t) = f(t, y)$, las derivadas sucesivas se pueden calcular usando regla de la cadena (en dos variables).

$$\begin{cases} y^{(0)}(t) &= f^{[0]}(t, y) = f(t, y), \\ y^{(k+1)}(t) &= f^{[k+1]}(t, y) = f_{\color{red}t}^{[k]}(t, y) + f_{\color{red}y}^{[k]}(t, y) f(t, y), \quad k = 0, 1, 2, \dots, m. \end{cases}$$

Entonces, sacando el factor común h queda

$$y(t_{i+1}) \approx y_{i+1} = y_i + \color{red}h \left[f^{[0]}(t_i, y_i) + \frac{h}{2} f^{[1]}(t_i, y_i) + \frac{h^2}{3!} f^{[2]}(t_i, y_i) + \dots + \frac{h^{m-1}}{m!} f^{[m-1]}(t_i, y_i) \right], \quad i = 0, 1, \dots, n. \quad (8.4)$$

Observe que $f^{[k+1]}(t_i, y_i)$ se construye con las derivadas parciales “anteriores”:

$$f^{[k+1]}(t_i, y_i) = f_{\color{red}t}^{[k]}(t_i, y_i) + f_{\color{red}y}^{[k]}(t_i, y_i) f(t_i, y_i)$$

El siguiente ejemplo muestra el uso de esta fórmula.

Ejemplo 8.2

Consideremos el problema de valor inicial $\frac{dy}{dt} = 0.7y - t^2 + 1$, $t \in [1, 2]$, $y(1) = 1$. La solución exacta es $y(t) = 1.42857t^2 + 4.08163t - 4.42583e^{0.7t} + 4.40233$.

Vamos a aplicar el método de Taylor de orden $m = 4$ con $n = 10$. Tenemos $a = 1$, $b = 2$, $h = 0.1$ y $t_i = 1 + 0.1i$. Ahora debemos calcular las derivadas,

$$f^{[0]}(t, y) = 0.7y - t^2 + 1,$$

$$f^{[1]}(t, y) = f_t^{[0]}(t, y) + f_y^{[0]}(t, y) \quad f(t, y) = -2t + 0.7(0.7y - t^2 + 1),$$

$$f^{[2]}(t, y) = f_t^{[1]}(t, y) + f_y^{[1]}(t, y) \quad f(t, y) = -2 - 2 \cdot 0.7^1 t + 0.7^2 (0.7y - t^2 + 1),$$

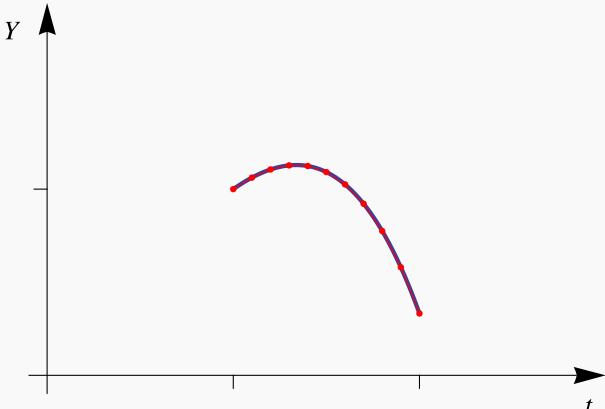
$$f^{[3]}(t, y) = f_t^{[2]}(t, y) + f_y^{[2]}(t, y) \quad f(t, y) = -2 \cdot 0.7^1 - 2 \cdot 0.7^2 t + 0.7^3 (0.7y - t^2 + 1).$$

$$\begin{cases} y_0 = 1 \\ y_{i+1} = y_i + h \left[f^{[0]}(t_i, y_i) + \frac{h}{2} f^{[1]}(t_i, y_i) + \frac{h^2}{3!} f^{[2]}(t_i, y_i) + \frac{h^3}{4!} f^{[3]}(t_i, y_i) \right], i = 0, 1, \dots, n. \end{cases}$$

$$\text{Así, } y_1 = y_0 + 0.1 \left[f^{[0]}(t_0, y_0) + \frac{0.1}{2} f^{[1]}(t_0, y_0) + \frac{0.1^2}{3!} f^{[2]}(t_0, y_0) + \frac{0.1^3}{4!} f^{[3]}(t_0, y_0) \right] = 1.06193158375.$$

En la tabla se continúa el cálculo hasta y_{10} .

Método de Taylor de orden 4		Valor exacto
t_i	y_i	$y(t_i)$
1.0	1.0	1.0
1.1	1.06193158375	1.061931432036279
1.2	1.105577521330614	1.105577223160223
1.3	1.127540292137498	1.127539827069448
1.4	1.124176027574661	1.124175372973675
1.5	1.091576648813122	1.091575779638891
1.6	1.025550709391196	1.025549597968614
1.7	0.92160284874683	0.92160146451555
1.8	0.77491175596324	0.77491006520478
1.9	0.58030653570613	0.58030450124654
2.0	0.33224136049833	0.33223894138437



Ejemplo 8.3

Considere el problema de valor inicial $y' = -40y + t e^{3t}$, $t \in [1, 2]$, $y(1) = 10$.

(a.) Aplique el *método de Taylor* de orden $m = 4$ para aproximar $y(1.2)$ usando $h = 0.2$.

Solución:

$$t_0 = 1, \quad y_0 = 10 \quad y \quad h = 0.2$$

$$f^{[0]}(t, y) = -40y + t e^{3t}, \quad f^{[0]}(1, 10) = -40 \cdot 10 + 1 \cdot e^{3 \cdot 1} = -379.9144630$$

$$f^{[1]}(t, y) = e^{3t}(1 - 37t) + 1600y$$

$$f^{[2]}(t, y) = e^{3t}(1489t - 34) - 64000y$$

$$f^{[3]}(t, y) = e^{3t}(1387 - 59533t) + 2560000y$$

$$\begin{aligned} y_1 &= y_0 + 0.2 \left[f^{[0]}(t_0, y_0) + \frac{0.2}{2} f^{[1]}(t_0, y_0) + \frac{0.2^2}{3!} f^{[2]}(t_0, y_0) + \frac{0.2^3}{4!} f^{[3]}(t_0, y_0) \right] \\ &= y_0 + 0.2 \left[-379.9144630 + \frac{0.2}{2} \cdot 15276.92067 + \frac{0.2^2}{3!} \cdot -610775.543 + \frac{0.2^3}{4!} \cdot 24432106.37 \right] \\ &\approx 1053.9952 \end{aligned}$$

(b.) Aplique el *método de Runge-Kutta* de orden 4 para aproximar $y(1.2)$ usando $h = 0.2$.

Solución: $t_0 = 1, \quad y_0 = 10 \quad y \quad h = 0.2$

$$k_1 = \frac{h}{2} f(t_0, y_0) = \frac{0.2}{2} \cdot -379.9144630768124 = -37.99144630768124$$

$$k_2 = \frac{h}{2} f\left(t_0 + \frac{h}{2}, y_0 + k_1\right) = \frac{0.2}{2} f\left(1 + \frac{0.2}{2}, 10 - 37.99...\right) = 114.9481755119973$$

$$k_3 = \frac{h}{2} f\left(t_0 + \frac{h}{2}, y_0 + k_2\right) = -496.8103117667169$$

$$k_4 = \frac{h}{2} f(t_0 + h, y_0 + 2k_3) = 3938.87428226697$$

$$y_1 = y_0 + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4) \approx 1055.7195211$$

Nota: La solución de este tipo de ecuación es de carácter exponencial. Se tomó $h = 0.2$ solo por simplicidad, pero un paso $h = 0.2$ es muy groso y nos da aproximaciones con errores muy grandes. Si usamos un h más

pequeño podemos obtener aproximaciones con más sentido, por ejemplo si $h = 0.0001$ obtenemos, en la iteración $i = 2000$, una aproximación muy cercana al valor correcto: $y(1.2) \approx y_{2000} = 1.004754122804870$. Nótese la gran diferencia obtenida con $h = 0.2$ y la obtenida con $h = 0.0001$

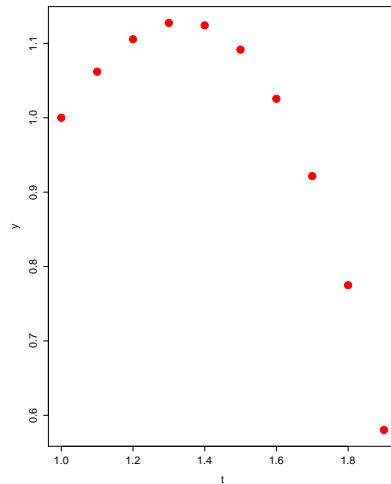
8.3.1 implementación

Para calcular las derivadas parciales vamos a usar el paquete **Deriv**. La función **Deriv(f, "t")** de este paquete, devuelve la derivada parcial de f respecto a t .

■ Código R 8.2: Método de Taylor (orden 4)

```
#install.packages(Deriv)
require(Deriv) # derivadas parciales
#--- Método de Taylor, orden 4
mtaylor4= function(f, t0, y0, h, n){
  #Datos igualmente espaciados iniciando en t0 = a, paso h. "n" datos
  t = seq(t0, t0 + (n-1)*h, by = h) # n datos
  y = rep(NA, times=n) # n datos
  y[1] = y0
  # Derivadas parciales con el paquete Deriv. Deriv(f)
  ft=Deriv(f, "t"); fy=Deriv(f, "y")
  f1 = function(t,y)
  ft(t,y)+fy(t,y)*f(t,y)
  f1t=Deriv(f1, "t");   f1y=Deriv(f1, "y")
  f2= function(t,y) f1t(t,y)+f1y(t,y)*f(t,y)
  f2t=Deriv(f2, "t");   f2y=Deriv(f2, "y")
  f3= function(t,y) f2t(t,y)+f2y(t,y)*f(t,y)    # orden m = 4
  for(i in 2:n ){
    f0i = f(t[i-1], y[i-1])
    f1i = f1(t[i-1], y[i-1])
    f2i = f2(t[i-1], y[i-1])
    f3i = f3(t[i-1], y[i-1])
    y[i] = y[i-1] + h*(f0i + h/2*f1i + h^2/6*f2i + h^3/24*f3i )
  }
  print(cbind(t,y)) #imprimir
  plot(t,y, pch=19, col="red",cex = 2) #gráfica
}
```

```
# --- Pruebas
f = function(t,y) 0.7*y - t^2 + 1
t0 = 1; y0 = 1; h = 0.1; n=10
mtaylor4(f, t0, y0, h, n)
# t           y
# [1,] 1.0 1.000000000000000
# [2,] 1.1 1.061927762500000
# [3,] 1.2 1.105569324685309
# [4,] 1.3 1.127527105681314
# [5,] 1.4 1.124157170794955
# [6,] 1.5 1.091551368745270
# ...
```



8.4 Métodos de Runge-Kutta.

Como decíamos, los métodos de un solo paso tienen la forma

$$y_{i+1} = y_i + h \cdot \Phi(t_i, y_i; h), \quad h > 0.$$

En el método de Euler la *función incremento* es

$$\Phi(t_i, y_i; h) = f(t_i, y_i)$$

Para el método de Taylor de orden 2 es,

$$\Phi(t_i, y_i; h) = f(t_i, y_i) + \frac{h}{2} [f_t(t_i, y_i) + f_y(t_i, y_i) f(t_i, y_i)] \quad (8.5)$$

Los métodos de Runge-Kutta son métodos diseñados pensando en imitar las expansiones de Taylor pero usando solo evaluaciones de la función $f(t, y)$. En el caso del método de Runge-Kutta de orden 2, se trata de modificar el método de Euler escribiendo

$$\Phi(t_i, y_i; h) = a_1 k_1 + a_2 k_2, \quad (8.6)$$

con $k_1 = f(t, y)$ y $k_2 = f(t + \alpha h, y + \beta h k_1)$, es decir, no se va a evaluar en la tangente hasta $t_i + h$ sino antes, usando la pendiente de la tangente en $(t + \alpha h, y + \beta h k_1)$.

Expandiendo 8.5 y 8.6 en potencias de h (usando la fórmula de Taylor en dos variables) y comparando se obtiene, entre varias opciones, $\alpha = 1$, $\beta = 1$, $a_1 = a_2 = 1/2$. Esto nos da un método Runge-Kutta de orden 2,

$$y_{i+1} = y_i + \frac{h}{2} [f(t_i, y_i) + f(t_i + h, y_i + h f(t_i, y_i))]$$

El método clásico de Runge-Kutta de orden 4 tiene una función de incremento que coincide con el polinomio de Taylor hasta el sumando con el término h^4 . Este método se puede escribir como,

$$\begin{cases} y_0 = y(a), \\ y_{i+1} = y_i + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4), \quad i = 0, 1, 2, \dots \end{cases}$$

donde k_1, k_2, k_3 y k_4 se calculan así:

$$\begin{aligned} k_1 &= \frac{h}{2}f(t_i, y_i), \\ k_2 &= \frac{h}{2}f\left(t_i + \frac{h}{2}, y_i + k_1\right), \\ k_3 &= \frac{h}{2}f\left(t_i + \frac{h}{2}, y_i + k_2\right), \\ k_4 &= \frac{h}{2}f(t_i + h, y_i + 2k_3), \end{aligned}$$

8.4.1 Algoritmo e implementación

El algoritmo es una copia de las fórmulas.

Implementación

■ Código R 8.3: Método de Runge-Kutta (orden 4)

```
rungekutta = function(f, t0, y0, h, n){
  t = seq(t0, t0+n*h, by=h)
  y = rep(NA, times=(n+1))
  # length(t)==length(y)
  y[1] = y0
  for(k in 2:(n+1)){
    k1=h/2*f(t[k-1],y[k-1])
    k2=h/2*f(t[k-1]+h/2, y[k-1]+k1)
    k3=h/2*f(t[k-1]+h/2, y[k-1]+k2)
    k4=h/2*f(t[k-1]+h, y[k-1]+2*k3)
    y[k] = y[k-1]+1/3*(k1+2*k2+2*k3+k4)
  }
  dat = cbind(t,y)
}
```

Algoritmo 8.2: Método de Runge-Kutta de orden 4

Datos: $f(t, y)$, m , a , b , y_0 , n

Salida: Imprime las aproximaciones (t_i, y_i) , $i = 0, 1, \dots, n$

```

1   $h = (b - a)/n;$ 
2   $t_0 = a;$ 
3  for  $i = 1$  ton do
4     $k_1 = \frac{h}{2}f(t_i, y_i);$ 
5     $k_2 = \frac{h}{2}f\left(t_i + \frac{h}{2}, y_i + k_1\right);$ 
6     $k_3 = \frac{h}{2}f\left(t_i + \frac{h}{2}, y_i + k_2\right);$ 
7     $k_4 = \frac{h}{2}f(t_i + h, y_i + 2k_3);$ 
8     $y_1 = y_i + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4);$ 
9     $t_0 = a + i \cdot h;$ 
10    $y_0 = y_1;$ 
11   print(( $t_0, y_0$ ));

```

```

print(as.matrix(dat))
plot(t,y,pch=20, col="red")
}

#Pruebas-----
options(digits = 15)
f=function(t,y) 0.7*y-t^2+1
t0=1; y0=1; h= 0.1; n= 10
rungekutta(f,t0,y0,h,n)
#---
#      t                  y
# [1,] 1.0 1.000000000000000
# [2,] 1.1 1.061931481666667
# [3,] 1.2 1.105577309762072
# [4,] 1.3 1.127539963145176
# [5,] 1.4 1.124175572644375
# [6,] 1.5 1.091576058813342
# ...

```

10

8.1 Considere el problema de valor inicial $y' = \cos(2t) + \sin(3t)$, $t \in [0, 1]$, $y(0) = 1$.

- a.) Usando el método de Euler, aproximar $y(0.4)$ con $h = 0.1$.
- b.) Usando el método de Taylor de orden 4, aproximar $y(0.4)$ con $h = 0.2$
- c.) Usando el método de Runge-Kutta de orden 4, aproximar $y(0.4)$ con $h = 0.2$

8.2 Considere el problema de valor inicial $y' = t \exp(3t) - 40y$, $t \in [1, 2]$, $y(1) = 10$.

- a.) Usando el método de Euler, aproximar $y(0.4)$ con $h = 0.1$.
- b.) Usando el método de Taylor de orden 4, aproximar $y(0.4)$ con $h = 0.2$
- c.) Usando el método de Runge-Kutta de orden 4, aproximar $y(0.4)$ con $h = 0.2$

8.3 Usando el método de Runge-Kutta de orden 4, aproximar $y(0.2)$ (con $h = 0.1$) si $y(t) = \int_0^t e^{-t^2} dt$. (Debe convertir el cálculo de la integral en un problema de valor inicial.)

8.4.2 Paquete “deSolve” de R

Las ecuaciones diferenciales (EDOs) están presentes en todo lugar en ciencias e ingeniería. **R** tiene varios paquetes para resolver numéricamente EDOs. Una visión general de los paquetes que se pueden usar se puede obtener en <https://cran.r-project.org/web/views/DifferentialEquations.html>.

Uno de los paquetes es **deSolve** En la descripción del paquete se puede leer

“deSolve provides functions that solve initial value problems of a system of first-order ordinary differential equations (ODE), of partial differential equations (PDE), of differential algebraic equations (DAE), and of delay differential equations. The functions provide an interface to the FORTRAN functions lsoda, lsodar, lsode, lsodes of the ODEPACK collection, to the FORTRAN functions dvode and daspk and a C-implementation of solvers of the Runge-Kutta family with fixed or variable time steps. The package contains routines designed for solving ODEs resulting from 1-D, 2-D and 3-D partial differential equations (PDE) that have been converted to ODEs by numerical differencing.”

Para resolver numéricamente ODEs podemos usar la función **ode()** del paquete **deSolve**.

```
ode(y, times, func, parms, method = c("lsoda",
  "lsode", "lsodes", "lsodar",
  "vode", "daspk", "euler", "rk4",
  "ode23", "ode45", "radau", "bdf",
  "bdf_d", "adams", "impAdams",
  "impAdams_d"), ...)
```

Como se ve, hay varios métodos que se pueden invocar. Si la función **func** no tiene parámetros, se podría poner **parms=NULL** cuando se invoca **ode()**.

Ejemplo 8.4 Termodinámica

Considere un cuerpo con temperatura interna T el cual se encuentra en un ambiente con temperatura constante T_e . Suponga que su masa m se concentrada en un solo punto. Entonces la transferencia de calor entre el cuerpo y el entorno externo puede ser descrita con la ley de Stefan-Boltzmann,

$$\nu(t) = \epsilon \gamma S (T^4(t) - T_e^4)$$

donde t es tiempo y ϵ es la constante de Boltzmann ($\epsilon = 5.6 \times 10^{-8} J/m^2 K^2 s$, donde J = joule, K = Kelvin y "m" y "s" son como usual, metros y segundos). γ es la constante de "emisividad" del cuerpo, S el área de la superficie y ν es la tasa de transferencia del calor. La tasa de variación de la energía $E(t) = mCT(t)$ (donde C indica el calor específico del material que constituye el cuerpo) es igual, en valor absoluto, a la tasa ν . En consecuencia, si $T(0) = T_0$, el cálculo de $T(t)$ requiere la solución de la ecuación diferencial ordinaria

$$\frac{dT}{dt} = -\frac{\nu(t)}{mC} \quad (*)$$

Usando 20 intervalos iguales y t variando de 0 a 200 segundos, resuelva numéricamente la ecuación (*) si el cuerpo es un cubo de lados de longitud 1 m y masa igual a 1 Kg. Asuma que $T_0 = 180 K$, $T_e = 200 K$, $\gamma = 0.5$ y $C = 100 J/(Kg/K)$. Hacer una representación gráfica del resultado.

Solución:

$S = 6m^2$. Sustituyendo los valores dados, el problema queda

$$\frac{dT}{dt} = -1.68 \times 10^{-9} * T(t)^4 + 2.6880 \text{ con } T(0) = 180 \text{ y } t \in [0, 200]$$

Usando la función **ode()** del paquete **deSolve** el código sería,

```
#install.packages("deSolve")
require(deSolve)
# La función ode() requiere obligatoriamente, varias cosas que debemos agregar
# ode(valores iniciales, tis , func, parms, method, ...)

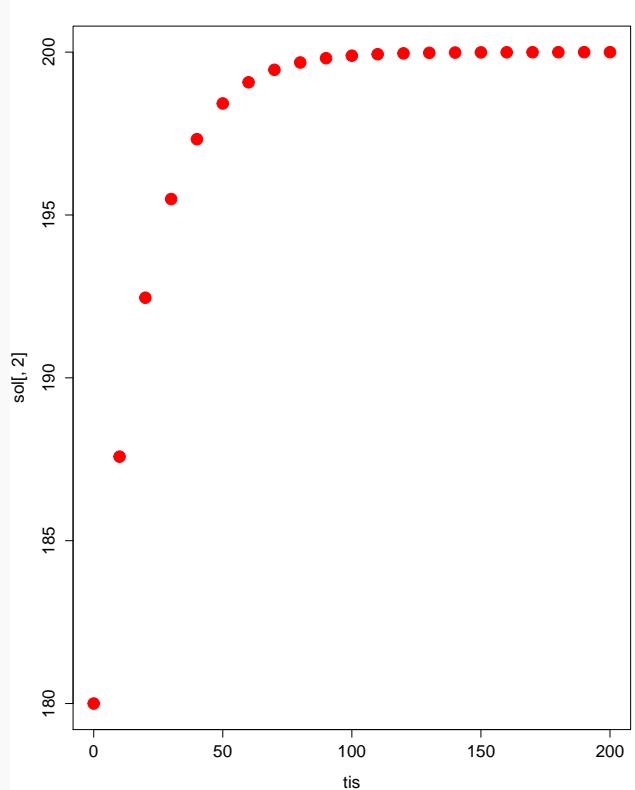
fp = function(t,y, parms){
  s = -1.68*10^(-9)*y^4+2.6880
  return(list(s)) # ode requiere salida sea una lista
}
```

```
tis= seq(0,200,200/20)
# Usamos la función ode()
sol = ode(c(180), tis, fp, parms=NULL, method = "rk4") # método Runge Kutta orden 4

# Salida
tabla = cbind(tis, sol[,2] )
colnames(tabla) = c("ti", "Ti")
tabla

# Representación
plot(tis, sol[,2] )
```

```
# ----- Salida -----
    ti      Ti
[1,] 0 180.0 #cond inicial T(0)=180
[2,] 10 187.580416562439
[3,] 20 192.460046001208
[4,] 30 195.489656884594
[5,] 40 197.326828018094
[6,] 50 198.424598570510
[7,] 60 199.074694193581
[8,] 70 199.457615740818
[9,] 80 199.682448224214
[10,] 90 199.814211064647
[11,] 100 199.891345429875
[12,] 110 199.936470940649
[13,] 120 199.962860490560
[14,] 130 199.978289770848
[15,] 140 199.987309699712
[16,] 150 199.992582334281
[17,] 160 199.995664336939
[18,] 170 199.997465807242
[19,] 180 199.998518773943
[20,] 190 199.999134231810
[21,] 200 199.999493964391
```



Ejemplo 8.5

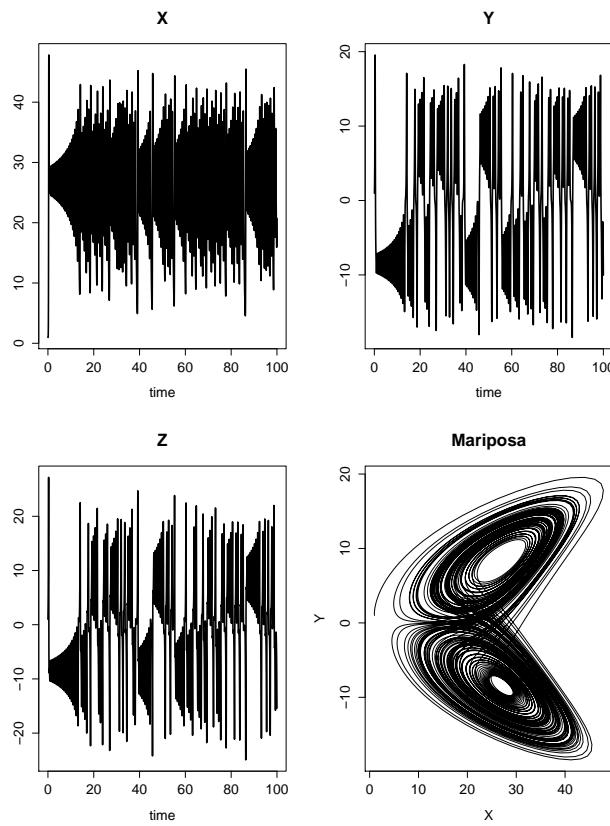
Las ecuaciones de Lorentz es un sistema dinámico con comportamiento caótico (el primero en ser descrito). ESte modelo consiste de tres ecuaciones diferenciales que expresan la dinámica de tres variables x, y, z que se asume, representan un comportamiento idealizado de la atmósfera de la tieraa. El modelo es

$$\begin{aligned}x' &= ax + yz \\y' &= b(y - z) \\z' &= -xy + cy - z\end{aligned}$$

Las varibales x, y y z representan la distribución horizontal y vertical de la temperatura y el flujo convectivo (la “convección es” la producción de flujos de gases y líquidos por el contacto con cuerpos u objetos de mayor temperatura) y $a = -8/3$, $b = -10$ y $c = 28$ son parámetros.

La solución numérica sería algo así:

```
a = -8/3; b = -10; c = 28
yini = c(X = 1, Y = 1, Z = 1)
Lorenz = function (t, y, parms) {
  with(as.list(y), {
    dX <- a * X + Y * Z
    dY <- b * (Y - Z)
    dZ <- -X * Y + c * Y - Z
    list(c(dX, dY, dZ))
  })
}
# Resolvemos para 100 días produciendo una salida cada 0.01 día
times = seq(from = 0, to = 100, by = 0.01)
out = ode(y = yini, times = times, func = Lorenz,parms = NULL)
# Gráfica
plot(out, lwd = 2)
plot(out[, "X"], out[, "Y"], type = "l", xlab = "X", ylab = "Y", main = "Mariposa")
```



8.5 Algunos Detalles Teóricos.

Definición 8.1

Consideremos un conjunto $D \subseteq \mathbb{R}^2$ y una función $f(t, y)$ definida en D . Si existe una constante $L > 0$ tal que

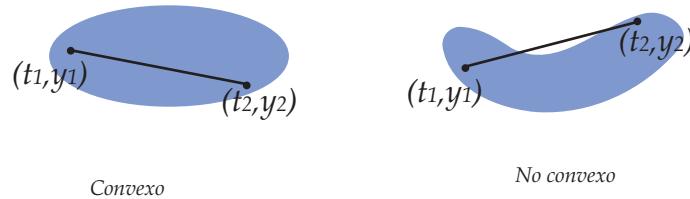
$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|, \quad \forall (t, y_1), (t, y_2) \in D$$

se dice que $f(t, y)$ cumple una condición de Lipschitz en la variable y en D . A L se le llama constante de Lipschitz para f .

Nota: Una condición *suficiente* para que $f(t, y)$ cumpla una condición de Lipschitz en D es que exista $L > 0$ tal que $\left| \frac{\partial f(t, y)}{\partial y} \right| \leq L, \quad \forall (t, y) \in D$

Definición 8.2

Un conjunto $D \subseteq \mathbb{R}$ se dice convexo si $\forall (t_1, y_1), (t_2, y_2) \in D$, el segmento $\{(1 - \lambda)(t_1, y_1) + \lambda(t_2, y_2), \lambda \in [0, 1]\}$ está contenido en D .



Nota: Observe que, cuando $\lambda = 0$ estamos en el punto inicial (t_1, y_1) y cuando $\lambda = 1$ estamos en el punto final (t_2, y_2) . $\lambda = 1/2$ corresponde al punto medio del segmento.

Teorema 8.2

Si $D = \{(t, y) : a \leq t \leq b, -\infty \leq y \leq \infty\}$ y si $f(t, y)$ es continua en D y satisface una condición de Lipschitz respecto a y en D , entonces el problema (*) tiene una solución única $y(t)$ para $a \leq t \leq b$.

Nota: Un problema de valor inicial está *bien planteado* si pequeños cambios o perturbaciones en el planteo del problema (debido a errores de redondeo en el problema inicial, por ejemplo), ocasiona cambios pequeños en la solución del problema. Si un problema cumple las hipótesis del teorema anterior, entonces está bien planteado.

Ejemplo 8.6

Consideremos el problema de valor inicial $y' = y - t^2 + 1$, $t \in [0, 4]$, $y(0) = 0.5$. Aquí $f(t, y) = y - t^2 + 1$. Si $D = \{(t, y) : 0 \leq t \leq 4, -\infty \leq y \leq \infty\}$, entonces como

$$\left| \frac{\partial f(t, y)}{\partial y} \right| = |1| \quad \forall (t, y) \in D$$

f cumple una condición de Lipschitz en y (en este caso podemos tomar $L = 1$). Además, como $f(t, y)$ es continua en D , el problema de valor inicial tiene una solución única. De hecho la única solución es $y(t) = (t + 1)^2 - 0.5e^t$

8.6 Estimación del error

Teorema 8.3

Si $D = \{(t, y) : a \leq t \leq b, -\infty \leq y \leq \infty\}$ y si $f(t, y)$ es continua en D y satisface una condición de Lipschitz respecto a y en D con constante L entonces si existe una constante M tal que

$$|y''(t)| \leq M, \quad \forall t \in [a, b]$$

entonces para cada $i = 0, 1, 2, \dots, n$,

$$|y(t_i) - y_i| \leq \frac{hM}{2L} (e^{L(t_i-a)} - 1)$$

Nota: para calcular $|y''(t)|$ usamos regla de la cadena: $y''(t) = \frac{\partial f}{\partial t}(t, y) + \frac{\partial f}{\partial y}(t, y) \cdot y'(t)$. Posiblemente sea difícil obtener M dado que puede ser necesaria información acerca de $y(t)$.



Revisado: Marzo, 2016

Versión actualizada de este libro:

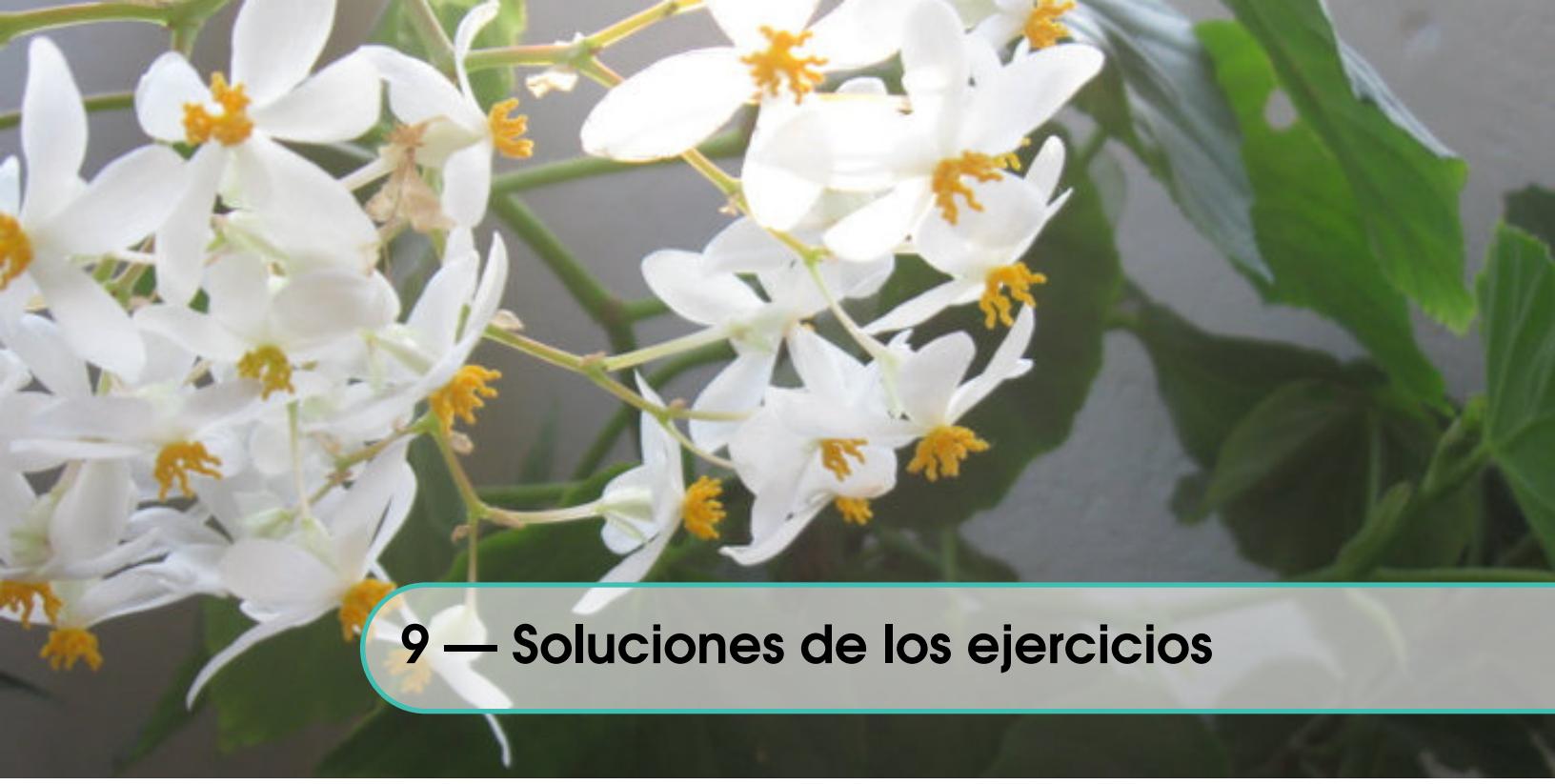
<https://tecdigital.tec.ac.cr/revistamatematica/Libros/>



Bibliografía

- [1] W. Gautschi. *Numerical Analysis. An Introduction.* Birkhäuser, 1997.
- [2] P. Henrici. *Essentials of Numerical Analysis.* Wiley, New York, 1982.
- [3] J. Stoer,. *Introduction to Numerical Analysis.* 3rd ed. Springer, 2002.
- [4] D. Kahaner, K. Moler, S. Nash. *Numerical Methods and Software.* Prentice Hall.1989.
- [5] D. Kincaid, W. Cheney. *Numerical Analysis. Mathematics of Scientific Computing.* Brooks-Cole Publishing Co.1991.
- [6] R. Burden, J. Faires. *AnAlisis NumArico.* 6ta ed. Thomson. 1998.
- [7] E. Cheney, *Introduction to Approximation Theory.* Internat. Ser. Pure ans Applied Mathematics. McGraw-Hill. 1966.
- [8] J.P. Berrut, L. N. Trefethen. “Barycentric Lagrange Interpolation” Siam Rewiew. Vol. 46, No. 3. 2004.
- [9] J. Higham, “The numerical stability of barycentric Lagrange interpolation”. IMA Journal of Numerical Analysis 24. 2004.
- [10] Chapra, S.; Canale, R. *Métodos Numéricos para Ingenieros.* Ed. Mc Graw Hill, 4a. ed., 2002.
- [11] Mathews, J; Fink, K.*Métodos Numéricos con MATLAB.* Prentice Hall, 3a. ed., 2000.
- [12] Dahlquist, G. Björk, A. *Numerical Mathematics in Scientific Computation.* www.mai.liu.se/~akbjol. Consultada en Mayo, 2006.
- [13] Ralston, A.; Rabinowitz, P. *A First Course in Numerical Analysis.* 2nd ed. Dover, 1978.
- [14] “CRAN Task View: Numerical Mathematics”. <http://cran.r-project.org/web/views/NumericalMathematics.html>

- [15] "Data Analysts Captivated by R's Power". <http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?pagewanted=all>
- [16] W. Mora, A. Borbón. *Edición de Textos Científicos con LaTeX*. ITCR. <http://tecdigital.tec.ac.cr/revistamatematica/Libros/>
- [17] Norman Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press; 1 edition (October 15, 2011).



9 — Soluciones de los ejercicios

Soluciones del Capítulo 1

Soluciones del Capítulo 2

2.1 $|p - \tilde{p}| \leq 0.5 \times 10^{-4}$ pero $|p - \tilde{p}| > 0.5 \times 10^{-5}$ así que tenemos 4 decimales correctos. Luego, solo 0.001 y 0.0002 son $> 10^{-4}$ así que solo hay dos cifras significativas. solo

2.2 $|p - \tilde{p}| \leq 0.000001 \leq 0.5 \times 10^{-5}$ pero no se sabe si $|p - \tilde{p}| < 10^{-6}$, entonces solo podemos asegurar 5 decimales correctos.

2.3 Sirve cualquier $\delta \leq 0.5 \times 10^{-3}$.

Soluciones del Capítulo 3

Soluciones del Capítulo 4

4.1

a.)

b.)

c.)

d.)

e.)

4.2

a.)

b.)

4.3

4.4

a.)

b.)

4.5

4.6

4.7

4.8

4.9

4.10

a.)

b.)

4.11

4.12

4.13

4.14

4.15

4.16

4.17

4.18

4.19

a.)

b.)

4.20

4.21

4.22

4.12

4.13

4.14

d.)

4.23

4.15

Soluciones del Capítulo 5

Soluciones del Capítulo 6

6.1

a.)

$$\begin{aligned} P_3(x) &= \frac{1 \cdot (x-1)(x-3)(x-4)}{(0-1)(0-3)(0-4)} \\ &+ \frac{2 \cdot (x-0)(x-3)(x-4)}{(1-0)(1-3)(1-4)} \\ &+ \frac{0 \cdot (x-0)(x-1)(x-4)}{(3-0)(3-1)(3-4)} \\ &+ \frac{4 \cdot (x-0)(x-1)(x-3)}{(4-0)(4-1)(4-3)} \end{aligned}$$

b.)

c.) $f(3.5) \approx P_3(3.5) = 1.21875$

6.2

a.) $P_3(x) = x(x-1)(x-3)(x-4) \left(\frac{1}{3(x-1)} - \frac{1}{12x} + \frac{1}{3(x-4)} \right)$

b.) $P_3(x) = \frac{\frac{1}{3(x-1)} - \frac{1}{12x} + \frac{1}{3(x-4)}}{-\frac{1}{6(x-3)} + \frac{1}{6(x-1)} - \frac{1}{12x} + \frac{1}{12(x-4)}}$

c.) $P(0) = 1$, etc.

d.) $f(3.5) \approx P_3(3.5) = 1.21875$

6.3 Forma Modificada,

$$P_3(x) = (x - 0.5)(x - 0.4)(x - 0.3)(x - 0.2) \left(-\frac{4700.}{x - 0.4} + \frac{2650.}{x - 0.3} - \frac{200.}{x - 0.2} + \frac{1750.}{x - 0.5} \right)$$

Forma Baricéntrica,

$$P_3(x) = \frac{-\frac{4700.}{x - 0.4} + \frac{2650.}{x - 0.3} - \frac{200.}{x - 0.2} + \frac{1750.}{x - 0.5}}{-\frac{500.}{x - 0.4} + \frac{500.}{x - 0.3} - \frac{166.667}{x - 0.2} + \frac{166.667}{x - 0.5}}$$

$$f(0.35) \approx P_3(0.35) = 7.5375$$

6.4

$$\begin{aligned} P_1(x) &= y_0 L_{n,0}(x) + y_1 L_{n,1}(x) \\ &= y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)} \\ &= \frac{y_0(x - x_1) - y_1 x + y_1 x_0 + \cancel{x_1 y_1} - \cancel{x_1 y_1}}{x_0 - x_1} \\ &= \frac{(y_0 - y_1)}{(x_0 - x_1)} (x - x_1) + y_1 \end{aligned}$$

6.5

a.) $P_2(x) = 44.875(x - 0.4)(x - 0.2) + 38.5x(x - 0.2) - 77.75(x - 0.4)x$

b.) $J_0(0.25) \approx P_3(0.25)/\pi = 0.974128$

6.6 La tabla acumulada es,

Salarios (\$)	1000	2000	3000	4000
Frecuencia	9	39	74	116

La cantidad estimada de personas con salario entre \$1000 y \$1500 es $P_3(1500) - 9 = 23.5 - 9 = 14.5$, es decir, unas 15 personas.

6.7 Como $1 + 3x^* = 1.75 \implies x^* = 0.25$. El polinomio interpolante es,

$$P_2(x) = 9.72544 \left(x - \frac{1}{3} \right) \left(x - \frac{1}{6} \right) - 7.49065x \left(x - \frac{1}{6} \right) - 2.54653 \left(x - \frac{1}{3} \right) x$$

$$\cos(1.75) \approx P_2(0.25) = -0.17054$$

6.8

a.) Usamos la subtabla,

$v \text{ (m}^3/\text{kg)}$	0.10377	0.11144
$s \text{ (kJ/Kg}\cdot\text{K)}$	6.4147	6.5453

El polinomio interpolante es $P_1(x) = -836.336(x - 0.11144) + 853.364(x - 0.10377)$. La entropía s para un volumen específico v de $0.108 \text{ m}^3/\text{kg}$ es $6.48673(\text{kJ/Kg}\cdot\text{K})$

b.) En este caso usamos los tres datos, el polinomio interpolante es $P_2(x) = 38665.6(x - 0.1254)(x - 0.11144) + 22408.7(x - 0.10377)(x - 0.11144) - 61129.2(x - 0.1254)(x - 0.10377)$. La entropía s para un volumen específico v de $0.108 \text{ m}^3/\text{kg}$ es $6.48753(\text{kJ/Kg}\cdot\text{K})$

6.9 Usando toda la tabla obtenemos $f(0.25) \approx P_5(0.25) = 0.00120042$.

6.10

$$|f(2.71) - P(2.71)| \leq \left| \frac{M}{3!} (2.71 - 1)(2.71 - 2)(2.71 - 3) \right|$$

$f^{(3)}(x) = 2/x$ y $f^{(4)}(x) = -2/x^2$. Como $f^{(4)}$ no se anula, $M = \max\{|f^{(3)}(1)|, |f^{(3)}(3)|\} = 2$.

$$|f(2.71) - P(2.71)| \leq 0.117363$$

6.11

$$|f(1.22) - P(1.22)| \leq \left| \frac{M}{4!} (1.22 - 0.5)(1.22 - 1.1)(1.22 - 1.2)(1.22 - 1.3) \right|$$

$$f^{(4)}(x) = -\frac{48(4x^4 - 12x^2 + 1)}{(2x^2 + 1)^4} \text{ y } f^{(5)}(x) = \frac{384(4x^5 - 20x^3 + 5x)}{(2x^2 + 1)^5}.$$

Los puntos críticos en $[0.5, 1.3]$ son $x = 0.513743$, $x = 0$. Entonces $M = 48$ y

$$|f(1.22) - P(1.22)| \leq 0.00027648$$

6.12

$$|e^{0.6} - P(0.6)| \leq \left| \frac{M}{3!} (0.6 - 0)(0.6 - 0.5)(0.6 - 1) \right|$$

$f^{(3)}(x) = f^{(4)}(x) = e^x$. No hay puntos críticos y $M = e$.

$$|e^{0.6} - P(0.6)| \leq 0.0108731$$

6.13 Aquí se pide estimar el error al interpolar $f(0.71) = \cos(3 \cdot 0.71 + 1)$ con $P(0.71)$. Si M es el máximo absoluto de $|f'''|$ en $[0, 1]$, entonces,

$$|f(0.71) - P(0.71)| \leq \left| \frac{M}{3!} (0.71 - 0)(0.71 - 0.5)(0.71 - 1) \right|$$

Máximo absoluto de $|f'''|$

Puntos críticos: $f^{(4)}(x) = 0 \implies 81 \cos(1 + 3x) = 0 \implies x = \frac{(2k+1)\frac{\pi}{2} - 1}{3}$, $k \in \mathbb{Z}$. Algunas soluciones son

$$\{..., -2.95133, -1.90413, -0.856932, 0.190265, 1.23746, 2.28466, 3.33186, ...\}.$$

El único punto crítico en el intervalo es $x = \frac{\frac{\pi}{2} - 1}{3} \approx 0.190265\dots$

Comparación: $M = \text{Máx}\{|f'''(0)|, |f'''\left(\frac{\frac{\pi}{2} - 1}{3}\right)|, |f'''(1)|\} = 27$. Así,

$$|f(0.71) - P(0.71)| \leq 0.194575$$

6.14

$$|f(0.25) - P(0.25)| \leq \left| \frac{M}{3!}(0.25 - 0)(0.25 - 1/6)(0.25 - 1/3) \right|$$

$f^{(3)}(x) = 27 \sin(1 + 3x)$, $f^{(4)}(x) = 81 \cos(1 + 3x)$. Punto crítico en $[0, 1/3]$, $x = 0.190265$. Entonces $M = 27$ y

$$|f(0.25) - P(0.25)| \leq 0.0078125$$

6.15 Aplicar la fórmula para el error general en interpolación cúbica $h = \pi/2$.

6.16 Aplicar la fórmula para el error general en interpolación cúbica con $h = 0.2$

6.17

6.18

6.19

6.20

$$|f(x) - P_n(x)| \leq \frac{M}{(n+1)!} |(x^* - x_0)(x^* - x_1) \cdots (x^* - x_n)| \leq \frac{M}{(n+1)!} (b-a)^{n+1}$$

pues $|(x^* - x_i)| \leq (b-a)$ para cada $i = 0, 1, \dots, n$.

6.21

$$\begin{cases} S_0(x) = 0 - 1/2(x-0) + 0(x-0)^2 + 3/2(x-0)^3 & \text{si } x \in [0, 1], \\ S_1(x) = 1 + 4(x-1) + 9/2(x-1)^2 - 3/2(x-1)^3 & \text{si } x \in [1, 2]. \end{cases}$$

6.22

a.)

$$\begin{cases} S_0(x) = -160 + 1.49061(x-100) + 0 \cdot (x-100)^2 - 0.00002(x-100)^3 & \text{si } x \in [100, 200], \\ S_1(x) = -35 + 0.76876(x-200) - 0.00721(x-200)^2 + 0.00002(x-200)^3 & \text{si } x \in [200, 300], \\ S_2(x) = -4.2 + 0.10832(x-300) + 0.0006(x-300)^2 - 3.77272 \times 10^{-6}(x-300)^3 & \text{si } x \in [300, 400], \\ S_3(x) = 9 + 0.117952153(x-400) - 0.0005(x-400)^2 + 1.28229 \times 10^{-6}(x-400)^3 & \text{si } x \in [400, 500], \\ S_4(x) = 16.9 + 0.05287(x-500) - 0.0001(x-500)^2 + 4.43540 \times 10^{-7}(x-500)^3 & \text{si } x \in [400, 500]. \end{cases}$$

b.) $S_3(450) = 13.8079 \text{ (cm}^3/\text{mol)}$

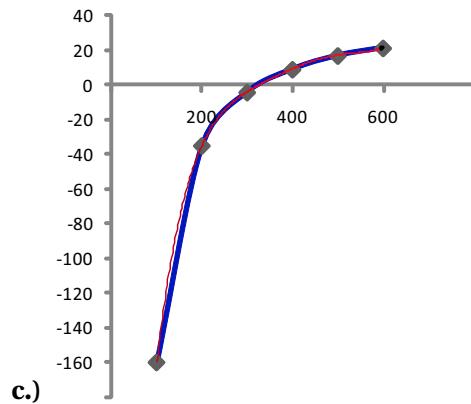


Tabla 9.1: El polinomio interpolante es la gráfica en roja, en azul está el trazador cúbico.

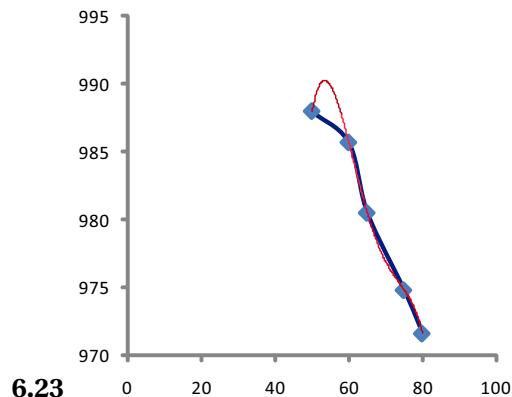


Figura 9.1: El polinomio interpolante es la gráfica en roja, en azul está el trazador cúbico.