Revista digital
Matemática, Educación e Internet
(http://tecdigital.tec.ac.cr/revistamatematica/).
Vol 16, No 1. Setiembre – Febrero 2016.
Artículo de sección
ISSN 1659 -0643

Cómo utilizar R en métodos numéricos

Walter Mora F.
wmora2@gmail.com
Escuela de Matemáticas
Instituto Tecnológico de Costa Rica

Resumen. R es ampliamente conocido como un lenguaje de programación y un entorno para análisis estadístico y la realización de gráficos de gran calidad. R es también un entorno de computación viable para la implementación y la aplicación de métodos numéricos de manera sencilla y efectiva. Aunque R permite varios estilos de programación, en al medida de lo posible en este artículo, se usa un estilo orientado a la "programación de arreglos" (llamado "vectorización"). Como R es interpretado, se incluye una sección mínima sobre cómo acelelar R usando "vectorización", o usando el paquete **Rcpp** para conectar R con **C++** y también paralelización.

Palabras clave: Método numéricos, lenguage R, álgebra lineal, ecuaciones no lineales, integración, ecuaciones diferenciales

Abstract. R is widely known as a programming language and a software environment for statistical analysis and high quality plotting. Additionally, R is also a software environment suitable for an effective and simple implementation of numerical methods. Although R allows several different programming styles, this article strives to use a style focused on "array programming" (called "vectorization"). Considering that R is "interpreted", an introductory section is included on how to accelerate R using "vectorization" or using **Rcpp** package to conect R with **C++** as well as parallelization.

KeyWords: Numerical methods, R language, linear algebra, non linear equations, integration, differential equations

1.1 Introducción

El propósito de este artículo es mostrar cómo se usa el lenguage R en la implementación de algoritmos en métodos numéricos a través de una pequeña excursión, implementado algoritmos o usando paquetes, por algunos tópicos usuales en los cursos de métodos numéricos. R es un lenguaje simple y eficiente, muy popular talvez por la facilidad con la que "se pueden convertir las ideas en código

de manera rápida y confiable". El lenguaje admite estilos de programación tradicionales, pero es más natural y eficiente si en vez de usar ciclos (`for, while, etc`) se "vectoriza" el código (en la medida de lo posible) porque en R "todo es un vector" y muchas de las funciones de R están implementadas en lenguages compilados como **C, C++** y **FORTRAN**. De esta manera, si vectorizamos, R solo debe interpretar el código y pasarlo a este otro código compilado para su ejecución.

Código R 1.1:Función no vectorizada

```
sumaF = function(k){
          s=0
          for(i in 1:k) s = s+1/i^2
          return(s)
}
```

Código R 1.2:Función vectorizada

```
# Versión vectorizada
sumaV = function(k){
             x = 1:k    #x=(1,2,...,k)
             sum(1/x^2)

}
```

En todo caso, para proyectos grandes, se pueden usar algunos paquetes para acelerar el código usando paralelización o cambiando funciones lentas por funciones implementadas en **C++**, por ejemplo.

En R hay muchos paquetes muy eficientes para usar en las tareas comunes del análisis numérico: Solución de sistemas de ecuaciones algebraicas lineales, ecuaciones no lineales, valores y vectores propios, interpolación y ajuste de curvas, diferenciación e integración numérica, optimización, solución ecuaciones diferenciales y ecuaciones diferenciales en derivadas parciales, análisis de Fourier, algo de análisis estadístico de datos, etc. Sin embargo, como es común, se debe implementar algoritmos por comprensión, para comunicarse con otros a través del lenguage de programación y sobre todo porque se debe modificar o implememtar nuevos programas para que encajen en un (nuevo) modelo o para hacer simulaciones especiales.

Este artículo incluye una introducción mínima al lenguage R (vectores y matrices), la implementación de algunos algoritmos (vectorizados en la medida que se pueda) de análisis numérico y una sección sobre cómo acelerar R .

1.2 Una introducción al lenguaje R

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 descaragar (Mac, Linux y Windows) en `http://www.rstudio.com/`. La instalación es directa en los tres casos.

1.2.1 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

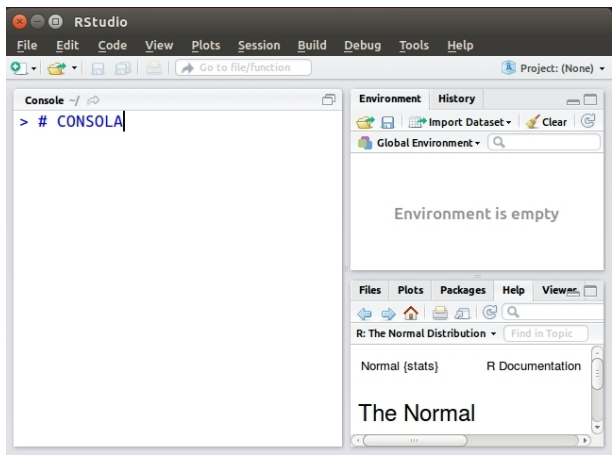


Figura 1.1: Sesión R inicial con RStudio

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.) **`Crt+Shift+C:`** Comentar código seleccionado

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`**.

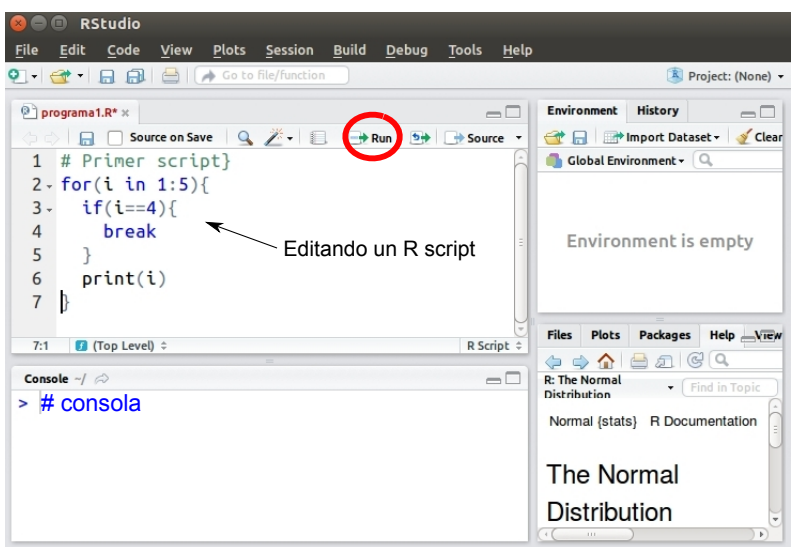


Figura 1.2: Editando un R script

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 seleccionando con el mouse) se presiona sobre el ícono **run** (para "lanzarlo a la consola")

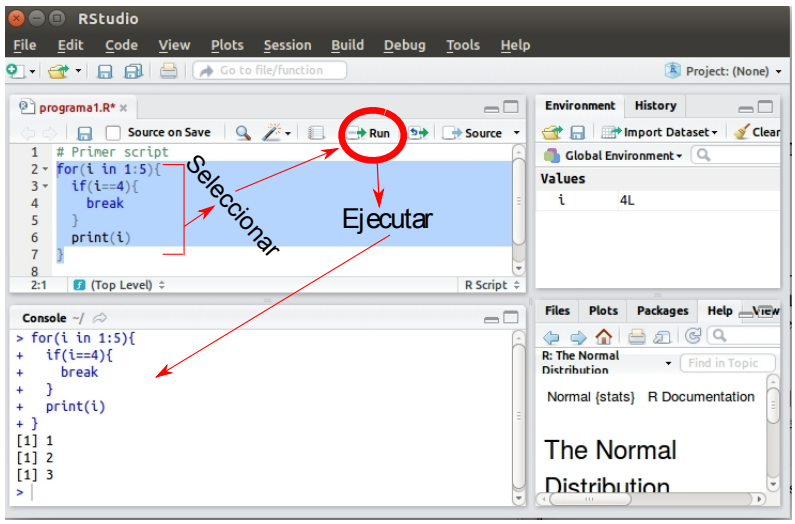
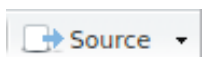


Figura 1.3: Ejecución del código seleccionado

Para ejecutar todo el código se usa **Crtl+Shift+Enter** o presionar el botón 

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).

R en la nube.

Hay varios sitios en Internet para usar R en la nube. Talvez la mejor manera sea en la página de SageMath (http://www.sagemath.org/). Después de registrase, 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.

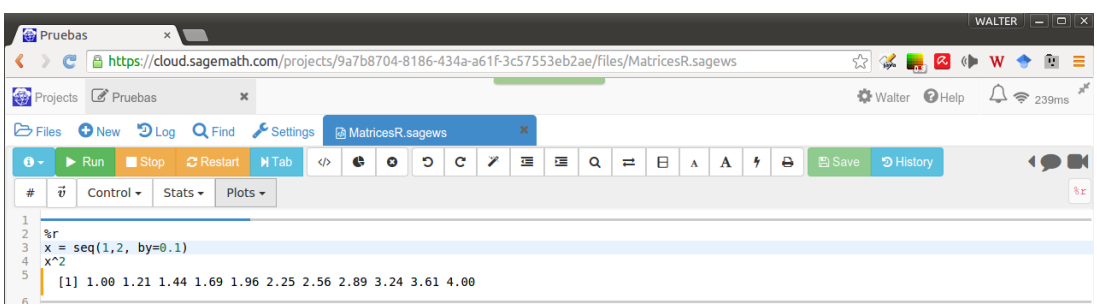


Figura 1.4: R en el entorno de SageMath en la nube

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

Figura 1.5: SageMath en Android

1.2.2 R como calculador

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

a.) Constantes: **`pi`** (π)

b.) Operadores: **`<,>, <=, >=, !=, !(Not), | (OR), &(And), == (comparar)`**

c.) Operaciones aritméticas: **`+,-,*,/, \^`** (potencias), **`%%`** (mod = resto de la divisón entera), y **`%/%`** (división entera).

d.) Logaritmos y exponenciales: **`log`** (logaritmo natural), **`log(x,b)`** ($\log_b x$) y **`exp(x)`** (e^x).

e.) Funcions trigonométricas: **`cos(x), sin(x), tan(x), acos(x), asin(x), atan(x), atan2(y,x)`** con **`x,y`** en radianes.

f.) Funciones misceláneas: **`abs(x), sqrt(x), floor(x), ceiling(x), max(x), sign(x)`**

1.2.3 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 "." y se puede definir una lista de variables separadas por ";".

■ Código R 1.3:Asignación de valores a las variables x0 y x1

```
x0 <-  0.143; x1 <- x0  # asignación con " <- " o con "="
x0 = (x0+x1)/x1^2
#[1] 13.98601
 label = "Tabla"        # string
```

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 1.4:"cat()"

```
x0 = 1
x1 = x0 - pi*x0 + 1
 cat("x0 =", x0, "\n", "x1 =", x1)  # "\n" = cambio de línea
# 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
```

1.2.4 Funciones

Las funciones se declaran usando la directiva `function(){... 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 1.5:Funciones

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

Un jemplo es la función discriminate, si $P(x) = ax^2 + bx + c$ entonces el discriminante es $\Delta = B^2 - 4ac$.

Código R 1.6:

```
d = function(a,b,c) b^2-4*a*c

# --- Pruebas
d(2,2,1)
#[1] -4
```

1.2.5 Vectores

Los vectores son el tipo básico da datos en R . 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)` (rep=repeat) y "`:`"

Código R 1.7: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)
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)
x  = rep(1, times  = 5)           # x = (1, 1, 1, 1, 1)
length(x)                         # 5
rep(x, 2)                         # (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
set.seed(1)                       # "semilla"
x = sample(1:10, 5)               # muestra seudoaleatoria de 5 números enteros de 1 a 10
x
#[1]  4 10  2  9  8               # Si cambia la semilla, cambia la muestra, sino no!
```

Operaciones algebraicas. A los vectores les podemos aplicar las operaciones algebraicas usuales: sumar, restar, multiplicar y dividir, exponenciación, etc. Todas las operaciones se hacen *miembro a miembro*.

■ Código R 1.8: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 1.9:

```
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 1.10:

```
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
```

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 1.11:**

```
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] 15
prod(x)
# [1] 120
mean(x)
# [1] 3
var(x)
#[1] 2.5
```

Ejemplo 1.1

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,

$$\begin{aligned} \sum_{i=1}^{k} \frac{1}{i^2} &= 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{k^2} \\ &= \texttt{sum(}1,\ 1/2^2,\ 1/3^2,\ \ldots,\ 1/k^2\ \texttt{)} \\ &\quad \text{Si } \texttt{x = 1:k} \text{ entonces} \\ &= \texttt{sum(1/x\^{}2)} \end{aligned}$$

■ Código R 1.12:

```
x = 1:100000
sum(1/x^2)
# [1] 1.644834
```

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[op]** donde **op** es un operación lógica válida.

■ Código R 1.13:

```
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
```

Ejemplo 1.2

Si tenemos un conjunto de datos numéricos **x = c(x1,x2,...,xn)** (una muestra) entonces su media (o promedio) es **sum(x)/lenght(x),** en R esta se calcula con **mean(x).**

Supongamos que tenemos un vector **x** de datos (una muestra) ordenados de menor a mayor, **x = (x1,x2,...,xn).** La $k-$ésima muestra podada es

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

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 1.14:Función k-ésima media podada

```
mediaP = function(x,k){
            n  = length(x)
            xs = sort(x)
            xp = xs[(k+1):(n-k)] #eliminar k primeros y los k últimos datos
            mean(xp)
}
```

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}.

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

■ Código R 1.15:k-ésima media Winsorizada

```
mediaW = function(x, k) {
            x = sort(x)
            n = length(x)
            x[1:k]       =  x[k+1] # inserta dato x[k+1] desde x[1] hasta x[k]
            x[(n-k+1):n] =  x[n-k] # inserta dato x[n-k] desde x[n-k+1] hasta x[n]
            mean(x)
}
```

Veamos las medias aplicadas a una muestra,

■ Código R 1.16: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 n 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 es la suma acumulada hasta 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 1.3

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, $e^x \approx 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^{20}}{20!}$. En particular,

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

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

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

$$\begin{aligned} e^x &\approx 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^k}{k!} \\ &\approx 1 + \texttt{sum(}\frac{x}{1}, \frac{x^2}{2!}, \frac{x^3}{3!}, ..., \frac{x^k}{k!}\texttt{)} \\ &\approx 1 + \texttt{sum(}\frac{x}{1}, \frac{x}{1}\cdot\frac{x}{2}, \frac{x}{1}\cdot\frac{x}{2}\cdot\frac{x}{3}, ..., \frac{x}{1}\cdot\frac{x}{2}\cdots\frac{x}{k}\texttt{)} \end{aligned}$$

como $\frac{x}{\texttt{1:k}} = (\frac{x}{1}, \frac{x}{2}, \frac{x}{3}, ..., \frac{x}{k})$, entonces

$$\approx 1 + \texttt{sum(cumprod(}\frac{x}{\texttt{1:k}}\texttt{))}$$

```
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, tiene problemas "de cancelación" (un problema asociado con la aritmética del computador); esta implementación solo es eficiente para números positivos.

1.2.6 Matrices y arreglos

Las matrices son arreglos bidimensionales y, por defecto, se declaran "por columnas" (como se ven las tablas de datos). 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(vector, nrow = ..., ncol = ...)
```

o también

```
A = matrix(vector, nrow = ..., ncol = ..., byrow = FALSE, dimnames = list(...,...)).
```

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 1.17: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
 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 = 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
 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, **`diag(A)`** construye una matriz diagonal o también extrae la diagonal de una matriz.

`I=diag(1, n)` es la matriz identidad $n \times n$ y **`D = diag(diag(A))`** es una matriz diagonal con la diagonal de la matriz A.

■ Código R 1.18:Función "diag()"

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

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

# Extraer la diagonal
  diag(A)
[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))
# diag(1, n, m) = Matriz nxm,  con 1's en las entradas a_{ii}
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
```

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 1.19: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 operacioes usuales de fila $\alpha F_i + \beta \overline{F}_j$ sobre la matriz A se hacen de manera natural

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

■ Código R 1.20: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 1.21: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
```

El *índice* de la entrada más grande, en valor absoluto, de la fila **k** de la matriz A es

```
which.max(abs(A[k, ]))
```

1.2.7 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 1.22: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
```

1.2.8 La 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 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 1.23: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 1.4 (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 elpromedio 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 1.5 (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(x0,x1,...,xn)** 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 1.24: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
```

1.2.9 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 1.6 (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)= NaN.` Una solución es separar el signo,

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

■ Código R 1.25: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 $xgeq0$, en otro caso nos queda una expresión indefinida.

■ Código R 1.26: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 1.27:

```
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 1.28: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 1.29:

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

Ejemplo 1.7

Consideremos una serie $\sum_{n=a}^{\infty} u_n$. Las "sumas parciales" son las sumas $S_k = \sum_{n=a}^{k} u_n$. Vamos a implementar una función que calcule sumas parciales.

■ Código R 1.30:Sumas parciales

```
sumasparciales = function(f, a,k){
  suma=0
  for(i in a:k) suma = suma+f(i)
  return(suma)
}
ui = function(i) 1/i^2
options(scipen=999)  # Deshabilitar notación científica
options(digits = 12) # doce dígitos
for(k in seq(500000,1000000, by = 100000)){
  cat(format(k, width=10),format(sumasparciales2(ui, 1, k),width=15),"\n",sep = "
        ")
}
#    500000       1.64493206685
#    600000       1.64493240018
#    700000       1.64493263828
#    800000       1.64493281685
#    900000       1.64493295574
```

```
#   1000000       1.64493306685

options(scipen=0)  # Habilitar notación científica
```

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 1.31:"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 1.32:while**

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

Por ejemplo,

■ **Código R 1.33:**

```
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 1.34:**

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

Por ejemplo,

■ **Código R 1.35:**

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

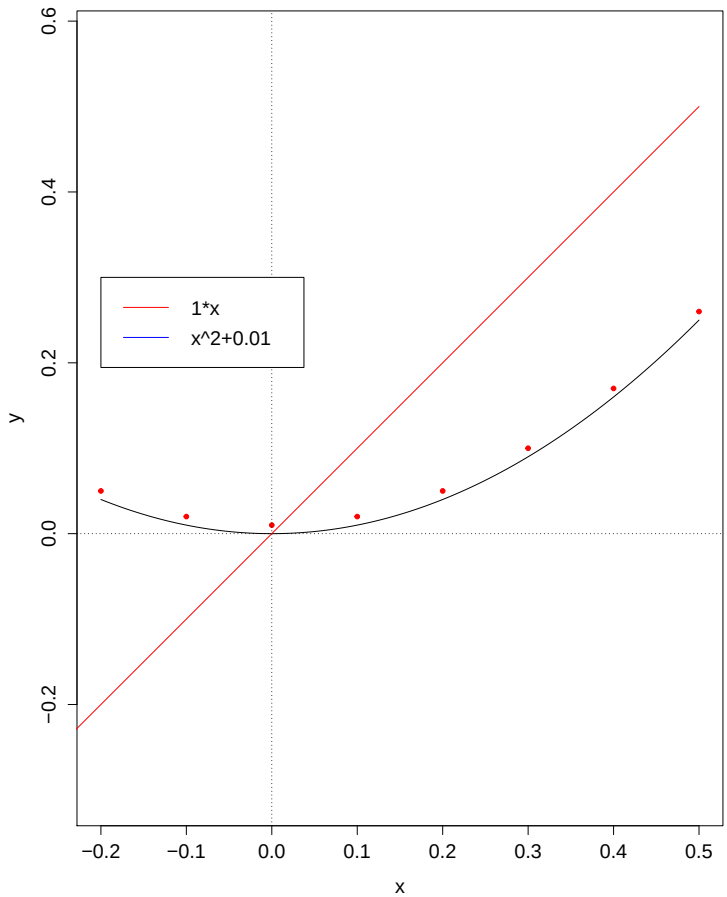
1.2.10 Gráficos

Podemos usar las funciones **plot()** y **curve()** para graficar funciones. También podemos usar la función **abline()** para agregar rectas adicionales.

■ **Código R 1.36:plot(), curve() y abline()**

```
x = seq(-0.2, 0.5, by=0.1)
y = x^2 + 0.01
# pares (x,y) en color rojo
plot(x,y, pch = 19, cex=0.7, col= "red", asp=1) # asp = 1: misma escala en ambos ejes
# Ejes X y Y con líneas punteadas
abline(h=0,v=0, lty=3)
```

```
curve(x^2, -0.2, 0.5, add=T)                    #add = T agrega el gráfico al anterior
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)
```



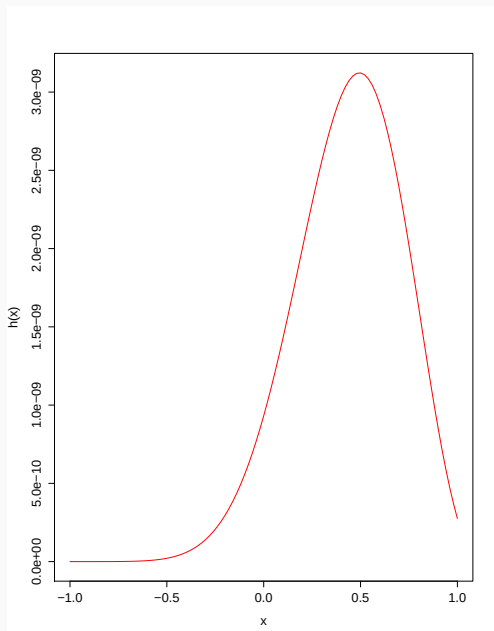
Con la función **curve()** se puede graficar funciones que recibe un vector (numérico) como entrada y que retorna un vector (numérico). Las funciones ordinarias de variable real se grafican de manera natural, pero las funciones que procesan vectores y devuelven un escalar requieren ser vectorizadas con la función **Vectorize().**

Ejemplo 1.8

Para implementar la representación gráfica de la función (de verosimilitud) $L(\alpha) = \prod_{i=1}^{n} \frac{1+\alpha \mathbf{x}_i}{2}$, donde **x** es un vector, se requiere usar la función **Vectorize**.

Código R 1.37: L requiere vectorización

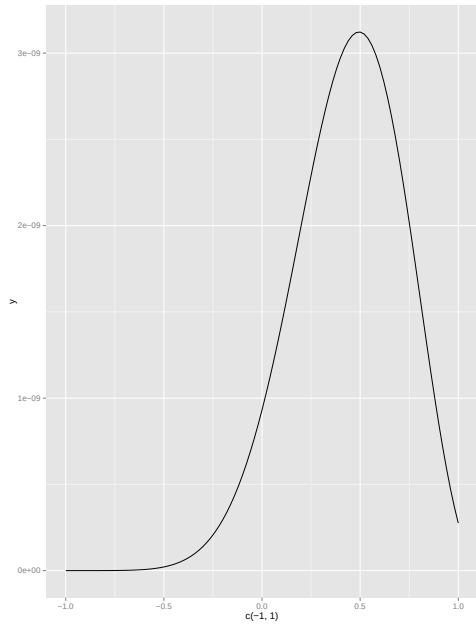
```
  x = c(0.41040018, 0.91061564,-0.61106896, 0.39736684, 0.37997637,
       0.34565436,0.01906680, -0.28765977, -0.33169289, 0.99989810, -0.35203164,
       0.10360470,0.30573300, 0.75283842, -0.33736278, -0.91455101, -0.76222116,
       0.27150040,-0.01257456, 0.68492778, -0.72343908, 0.45530570, 0.86249107,
       0.52578673,0.14145264, 0.76645754, -0.65536275, 0.12497668, 0.74971197,
       0.53839119)
L = function(alpha) prod(0.5*(1+alpha*x))
h = Vectorize(L)
curve(h,from =-1 , to = 1,  col = 2)
```



El paquete "ggplot2". Usando el paquete **ggplot2** se pueden obtener gráficas de mejor calidad.

Código R 1.38:L requiere vectorización

```
  x = c(0.41040018, ...) # anterior

L = function(alpha) prod(0.5*(1+alpha*x))
h = Vectorize(L)

library(ggplot2)
# c(-1,1) es el rango de x
qplot(c(-1,1), fun=h, stat="function",
                              geom="line")
```

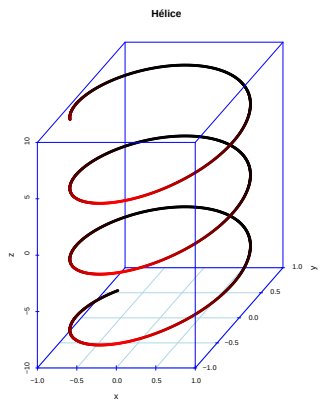


1.2.11 Gráficos 3D.

Hay varios paquetes en R para visualización en 3D. Por ejemplo el paquete **scatterplot3d**.

Código R 1.39:Paramterización 3D

```
require(scatterplot3d)
z <- seq(-10, 10, 0.01)
x <- cos(z)
y <- sin(z)
scatterplot3d(x, y, z, highlight.3d = TRUE,
                  col.axis = "blue",
                  col.grid = "lightblue",
                  main = "", pch = 20)
```



1.3 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.

En este caso, vamos a comparar el desempeño de funciones que calculan la suma parcial de una serie, con un ciclo **for** y su versión vectorizada. Usamos el paquete **rbenchmark** para medir el rendimiento en varias corridas.

■ Código R 1.40:Sumas parciales — comparación

```
sumaF= function(f, a,k){ suma=0
                         for(i in a:k) suma = suma+f(i)
                         return(suma)
}
# Versión vectorizada
sumaV = function(f, a,k){ i = a:k
                          sum(f(i))
}
# install.packages(rbenchmark) # instalar el paquete "rbenchmark"
library(rbenchmark)
k = 100000000         #k = 100 millones
f = function(n) 1/n^2
benchmark(sumaF(f, 1, k),sumaV(f, 1, k),replications = 5)[,c(1,3,4)]
#---
#             test    elapsed    relative
# 1 sumaF(f, 1, k)    339.322    55.21
# 2 sumaV(f, 1, k)    6.146      1.00
```

La comparación muestra como la versión vectorizada utilizó un acumulado de `6.146` segundos en las cinco corridas contra `339.322` segundos de la versión no vectorizada.

1.4 ¿Acelerar R?

R fue deliberadamente diseñado para hacer el análisis de datos y estadístico más fácil para el usuario. No fue diseñado para hacerlo veloz. Mientras que R es lento en comparación con otros lenguajes de programación, para la mayoría de los propósitos, es lo suficientemente rápido, aunque por suspuesto, un código mal escrito podría hacer que el desempeño sea pobre. La mayoría de los usuarios usa R solo para comprender datos y aunque es relativamente fácil hacer el código mucho más eficiente, no siempre es algo que un usuario valore como importante, excepto que tenga una motivación adicional.

En todo caso, para cálculos masivos, sería bueno tener una idea de algunas opciones para acelerar el código. Si no podemos vectorizar una función, hay algunos paquetes que nos ayudan a acelerar la ejecución. En lo que sigue vamos a usar algunos paquetes para este propósito con ejemplos simples (y sin entrar en todas las facetas de cada paquete).

1.4.1 El paquete Rcpp.

A veces solo necesitamos cambiar partes del código que son lentas por código altamente eficiente; el paquete **Rcpp** (Rc++) viene con funciones (así como clases de C++), que ofrecen una perfecta integración

de R y C++.

Los paquetes **RcppArmadillo**, **RcppEigen** habilitan de manera sencilla la integración de las librerías Armadillo y Eigen con R usando **Rcpp.** Las librerías Armadillo y Eigen son librerías C++ para aplicaciones del álgebra lineal.

Consideremos de nuevo las funciones para calcular la suma parcial de una serie,

■ Código R 1.41:Sumas parciales — comparación

```
sumaF= function(k){ suma=0
                        for(i in 1:k) suma = suma+1/i^2
                        return(suma)
}
# Versión vectorizada
sumaV = function(k){ x = 1:k
                     sum(1/x^2)
}
```

La versión vectorizada resulta ser inmensamente más eficiente que la versión no vectorizada, pero vamos usar el paquete **Rcpp** para mejorar aún más el desempeño. La función posiblemente más sencilla de usar en este paquete es la función **cppFunction().** Con esta función podemos incluir la versión en C++ de nuestra función, y utilizarla en el entorno R .

■ Código R 1.42:Sumas parciales — C++

```
#install.packages(Rcpp)
library(Rcpp)
# Función sumaCpp() n C++
cppFunction('double sumaCpp(long n){
            double s = 0;
            for(long i=1; i<=n; i++)
            {
            s = s + 1/(double)(i*i);
            }
            return s;
            }')
# Versión vectorizada de la misma función, en R
sumaV = function(k){ x = 1:k
                   sum(1/x^2)
}

# Usando la función sumaCpp()
sumaCpp(100000000)
# [1] 1.644934

# Comparación del desempeño-------------------------------------
# install.package(rbenchmark) # Instalar el paquete "rbenchmark"
```

```
library(rbenchmark)
k = 100000000                  # k = 100 millones
benchmark(sumaCpp(k), sumaV(k),replications = 5)[,c(1,3,4)]

#---
#        test   elapsed  relative
#1 sumaCpp(k)    2.192    1.000
#2   sumaV(k)    4.621    2.108
```

Como se ve en la tabla, en cinco corridas con $k = 100000000$, el desempeño de la función **sumaCpp()** es muy bueno comparado con la función vectorizada **sumaV().**

1.4.2 Paralelización.

Aunque en general más complicado, podemos programar en R pensando en "paralelo". Si nuestra PC tiene por ejemplo, cuatro nucleos podemos dividir el cálculo de la suma parcial de una serie (como el ejemplo anterior) en cuatro cálculos: Si vamos a sumar de $k = 1$ hasta $k = 100$ millones, podríamos hacer que:

a.) Procesador 1: suma los primeros 25 millones de términos,

b.) Procesador 2: suma los segundos 25 millones de términos,

c.) Procesador 3: suma los terceros 25 millones de términos,

d.) Procesador 4: suma los últimos 25 millones de términos.

Los cuatro procesos al mismo tiempo. Esto lo podemos hacer por ejemplo, usando el paquete **doParallel**

Lo primro que hay que hacer es registrar un "cluster" ("parallel backend"), es decir, el grupo de procesadores con lo que vamos a calcular; luego usamos la función **parLapply()** para *aplicar* una función **fun** (en este caso la fórmula para las sumas parciales) a cada uno de rangos de sumación.

La paralelización da buenos resultados para cálculos realmente grandes.

■ Código R 1.43:Sumas parciales — en paralelo

```
library(doParallel)
# Podemos saber cuántos clusters hay en la PC
# detectCores()  #[1] 4

#--- Creamos un cluster con 4 procesadores
micluster = makeCluster(4)
# --- registramos el cluster
registerDoParallel(micluster)

# Calculamos sum(1/x^2) para x = 1:k,  luego para x= (k+1):(2*k), etc. en paralelo
k=25000000
```

```
fun = function(x) sum(1/x^2)
s = parLapply(micluster, list(1:k, (k+1):(2*k), (2*k+1):(3*k), (3*k+1):(4*k)), fun )
#--- resultado de cada suma
s
# [[1]]
# [1] 1.644934

# [[2]]
# [1] 2e-08

# [[3]]
# [1] 6.666667e-09

# [[4]]
# [1] 3.333333e-09

# sumar todo
Reduce("+",s) # = s[[1]]+s[[2]]+s[[3]]+s[[4]]

#[1] 1.644934
```

Ahora podemos hacer una prueba de desempeño,

■ **Código R 1.44:Sumas parciales — desempeño**

```
fun = function(x) sum(1/x^2)
system.time(parLapply(micluster, list(1:k, (k+1):(2*k), (2*k+1):(3*k), (3*k+1):(4*k)), fun
    ))

#  user   system  elapsed
#  0.580   0.144   1.036

library(rbenchmark)
benchmark( parLapply(micluster, list(1:k, (k+1):(2*k), (2*k+1):(3*k), (3*k+1):(4*k)), fun )
    , replications = 5)[,c(3,4)]

#   elapsed relative
#1   4.982        1

stopCluster(micluster) # Es buena idea parar el proceso.
```

1.4.3 Paquete "plyr"

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 velocidad, 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").
Una alternativa para **apply()** (entre otras funciones) es usar la función equivalente **aaply()** del paquete **plyr.** Esta función tiene la opción de operar en paralelo, lo cual requiere crear un "cluster". A la fecha, para este paquete se requiere el paquete **doMC** para crear el cluster. Veamos un ejemplo: Vamos a calcular la media de cada una de las filas de una matriz 1000×100000.

■ Código R 1.45:Función "aaply()" del paquete plyr

```
# Usando plyr
library(doMC)
registerDoMC(4)     # Registrar un cluster de 4 procesadores
library(plyr)

m = 10000           # Este paquete es útil para grandes matrices
# Matriz A 1000x100000
A = matrix(rnorm(m * m), nrow = 1000)
# Desempeño de la función base apply()
system.time(apply(A, 1, mean))
# user  system elapsed
#  2.352   0.148   2.501

# Desempeño de la función aaply()
dim(A)
system.time(aaply(A, 1, mean, .parallel = TRUE))
# user  system elapsed
#  1.940   0.148   1.264
```

1.5 Ecuaciones no lineales.

La solución de ecuaciones es un tema central en análisis numérico. R tiene varias herramientas para la aproximación numérica de los ceros de una función.

Para aproximar los ceros de un polinomio se puede usar la función base **polyroot()** que usa el algoritmo de Jenkins−Traub (1972).

■ Código R 1.46:polyroot()

```
# Ceros de  P(x) = 1+x+x^3 (Observe el orden de los coeficientes)
curve(x^3+x+1 , -2, 2);abline(h=0, v=0, lty=3)
polyroot(c(1,1,0,1))

# [1]  0.3411639+1.161541i -0.6823278+0.000000i  0.3411639-1.161541i
# Observe que -0.6823278+0.000000i = -0.6823278
```

Para aproximar los ceros de una función f está la función **uniroot()** en la base de R . La documentación indica que esta función usa el método de Brent, este método es un método híbrido que combina bisección e interpolación cuadrática inversa.

```
uniroot(f, interval, ...,
       lower = min(interval), upper = max(interval),
       f.lower = f(lower, ...), f.upper = f(upper, ...),
       extendInt = c("no", "yes", "downX", "upX"), check.conv = FALSE,
       tol = .Machine$double.eps^0.25, maxiter = 1000, trace = 0)
```

■ Código R 1.47:uniroot()

```
# Ceros de f(x) = x - cos(x). Esta función tiene un cero en [0,1]
curve(x - cos(x) , -2, 2);abline(h=0, v=0, lty=3)
uniroot(f,  c(0,1), tol = 1e-15, maxiter=10 )

#$root
#[1] 0.7390851

#$f.root
#[1] 0

#$iter
#[1] 6

#$init.it
#[1] NA

#$estim.prec
#[1] 5.92507e-06
```

La función **uniroot.all** del paquete **rootSolve** trata de encontrar todos los ceros de una función en un intervalo

■ Código R 1.48:uniroot.all()

```
f = function(x) x^2 - x*cos(x)  # Tiene dos ceros en [0,1]
curve(f , -2, 2);abline(h=0, v=0, lty=3)

require(rootSolve)
ceros = uniroot.all(f, c(0,1), tol = 1e-15, maxiter=10)
ceros

# [1] 0.0000000 0.7390719
```

En el paquete **pracma** se pueden encontrar otros métodos clásicos como "bisección", "secante", "newton−raphson", etc. Sin embargo vamos a hacer la implementación de algunos métodos para tener una idea más completa de R .

1.5.1 El método de Punto Fijo. Animaciones

En el estudio de las poblaciones (por ejemplo de bacterias), la ecuación $x = xR(x)$ establece un vínculo entre el número de individuos x en una generación y el número de individuos en la siguiente generación. La función $R(x)$ modela la tasa de cambio de la población considerada.

Si $\phi(x) = xR(x)$, la dinámica de una población se define por el proceso iterativo

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

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 considera son las soluciones de un *problema de punto fijo*

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

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

$$\text{Aproximación inicial:} \quad x_0 \in \mathbb{R}. \qquad \text{Iteración de "punto fijo":} \quad x_{i+1} = g(x_i),\ i = 0,2,... \qquad (1.1)$$

El teorema que sigue da condiciones para la convergencia,

Teorema 1.1

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.

Criterio de parada.

Vamos a implementar una función **`puntofijo(g, x0, tol, maxIteraciones)`**. Para la implementación del método de punto fijo, se usa el criterio de parada

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

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

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

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.

Algoritmo e implementación. Vamos a implementar la función **`puntofijo(g, x0, tol, maxIter)`**, el criterio de parada que podríamos usar es

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

Algorithm 1.1: Iteración de Punto fijo.

Data: Una función continua g, x_0, **`tol`**, **`maxItr`**.
Result: Si hay convergencia, una aproximación x_n de un punto fijo.

1 $k = 0$;
2 **repeat**
3 $\quad x_1 = g(x_0)$;
4 $\quad dx = |x_1 - x_0|$;
5 $\quad x_0 = x_1$;
6 $\quad k = k + 1$;
7 **until** $dx \leq$ tol o $k >$ **`maxItr`** ;
8 **return** x_1

Código R 1.49:Iteración de "punto fijo"

```
#1e-9 = 0.000000001
puntofijo =function(g, x0, tol=1e-9, maxIter=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
```

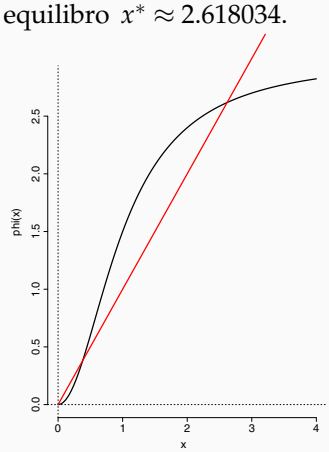
```
    #until
    if(dx< tol|| k > maxIter) break;
  }
  # Mensaje de salida
  if( dx > tol ){
    cat("No hubo convergencia   ")
    #return(NULL)
  } else{
    cat("x* es aproximadamente ", x1, " con error menor que ", tol)
  }
}
```

Ejemplo 1.9

Consideremos el modelo "predador-presa con saturación" con función $\phi(x) = \dfrac{rx^2}{1+(x/K)^2}$ donde $r = 3$ y $K = 1$.

De la gráfica se ve que hay un punto de equilibrio cercano a $x = 3$. Aplicamos la iteración de punto fijo con $x_0 = 2.5$, obtenemos la aproximación del punto de equilibro $x^* \approx 2.618034$.

```
phi = function(x) 3*x^2/(1+(x/1)^2)# r=3, K=1
curve(phi, 0, 4); abline(h=0, v=0, lty=3)
curve(1*x, 0,4, col="red", add=TRUE)
puntofijo(phi, 2.5, 1e-7, 15)
#x_ 1 =  2.586207
#...
#x_ 11 =  2.618034
#x_ 12 =  2.618034
#x* es aprox 2.618034  con error menor que  1e-07
```



1.5.2 Animación

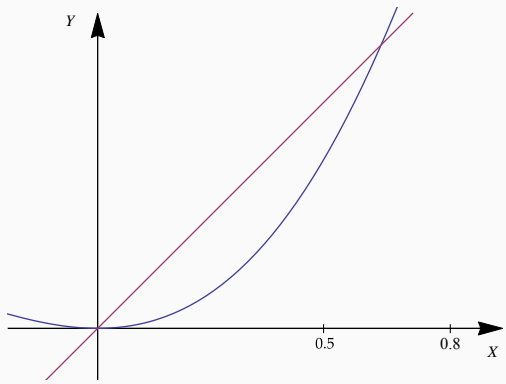
Podemos hacer una animación de un proceso iterativo usando el paquete **animation**. El proceso es simple: Creamos un ciclo con el gráfico de cada iteración y la animación es una sucesión de cuadros a cierta velocidad, posiblemente con pausas. Para generar la animación solo se requiere establecer el intervalo de tiempo con la función **ani.options()** y en el ciclo establecer las pausas con la función **ani.pause()**.

Ejemplo 1.10

La ecuación $(x+1)\operatorname{sen}(x^2) - x = 0$ 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

$$(x+1)\operatorname{sen}(x^2) = x$$

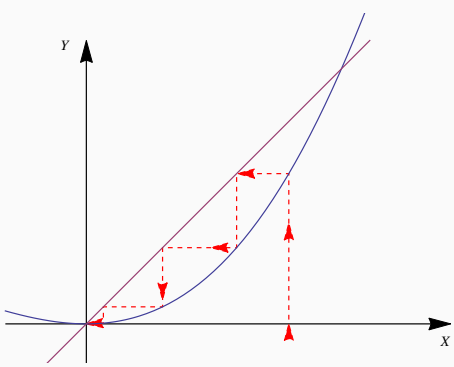
De acuerdo a la gráfica, hay dos puntos fijos, y uno de ellos x_1^*, está en $[0.5, 0.8]$. Por la forma de la gráfica, parece que el método de punto fijo no va a detectar a x_1^*.

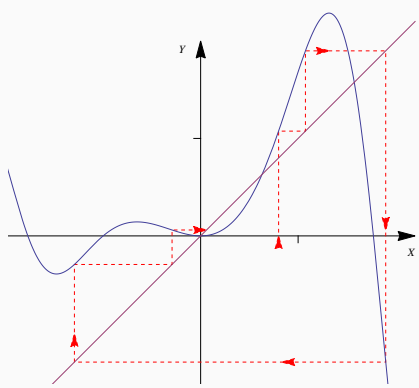


Corremos nuestro programa con valores **`x0 = 0.8`** y **`x0= 0.5`** y observamos que, en ambos casos, el método solo converge a $x_0^* = 0$.

```
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
```

Un animación gráfica del método nos puede muestrar con más detalle qué puede estar pasando.





Para implementar la animación hay que crear las flechas (**arrows**) una por una, como si fuera una secuencia de película. Para ver la animación (en la ventana d gráficos de **RStudio, por ejemplo**) solo debe correr el código que sigue (este código también se puede exportar a Internet como un video interactivo).

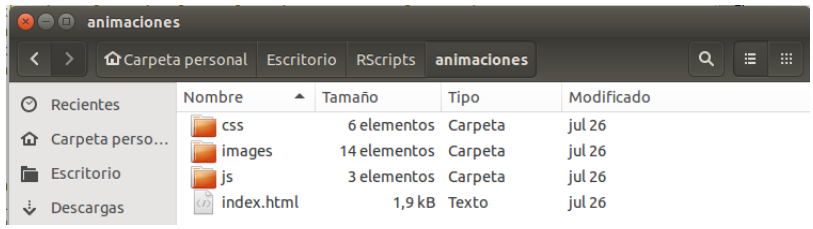
■ Código R 1.50:Animación de "punto fijo"

```
require(animation)
ani.options(interval=0.6) # tiempo entre cada cuadro, en segundos

puntofijoAnimacion = function(g, x0, numiter, a, b){ # [a,b] = intervalo del gráfico
  xa = x0
  xb = g(xa)
  iter = 1
  # Gráfico de fondo
  plot(g, a, b, col="blue") #asp=1
  curve((x), a, b, add=T); abline(h=0, v=0, lty=2)
  arrows(x0, 0, xa, xb, col="red", lty=1,length = 0.15, angle = 20)
  # Cuadros ("Frames")
  while(iter < numiter){
    arrows(xa, xb, xb, xb, col="red", lty=1,length = 0.15, angle = 20)
    ani.pause() # pausa
    arrows(xb, xb, xb, g(xb), col="red", lty=1,length = 0.15, angle = 20)
    ani.pause() # pausa
    xa   = xb
    xb   = g(xa)
    iter = iter+1
  }
}
#--- Correr la animación
g = function(x) (x+1)*sin(x^2)
puntofijoAnimacion(g, 0.8, 6, -3, 2)
```

Exportar la animación a Internet.

Se puede crear una página web con la animación anterior (y subirla a un servidor personal o uno gratuito como DropBox, por ejemplo) con la función **saveHTML()** del paquete **animate.** Al correr esta función, se abre inmediatamente el navegador predeterminado con la animación. Se puede usar el código del ejemplo anterior con cambios mínimos.

La función **saveHTML()** crea en la carpeta actual tres carpetas `css, images, js` y un archivo `index.html`. Estos elementos son los que hay que exportar para ver la animación en Internet.



Por ejemplo, la animación anterior se puede ver en

```
https://dl.dropboxusercontent.com/u/56645701/Rscripts/index.html
```

El código de la animación con la función **saveHTML()** es

Código R 1.51:saveHTML()

```
saveHTML({
 ani.options(interval=0.5, nmax = 14) # interval segundos
 #---código de la animación ----------------------------
 g = function(x) (x+1)*sin(x^2)
 pxa = rep(NA, times=ani.options("nmax")+1)
```

```
  pxb = rep(NA, times=ani.options("nmax")+1)
  x0 = 0.8
  a = -3
  b = 2
  xa = x0
  xb = g(x0)
  pxa[1]= x0
  pxb[1]=0
  pxa[2]= xa
  pxb[2]=xb
  k = 4
  # Los puntos de la animación
  while(k <= ani.options("nmax")){
    pxa[k-1]= xa
    pxa[k:(k+2)] = xb
    pxb[(k-1):(k+1)] = xb
    pxb[k+2] = g(xb)
    xa = xb
    xb = g(xa)
    k = k+2
  }
  # Hacer los frames
  for(j in 1: ani.options("nmax")){
    plot(g, a, b, col="blue") #asp=1
    curve((x), a, b, add=T); abline(h=0, v=0, lty=2)
    arrows(pxa[1:j], pxb[1:j], pxa[2:(j+1)], pxb[2:(j+1)], col="red", length = 0.15, angle
        =20)
    ani.pause() # pausa: paquete "animation"
  }
}, img.name = "puntofijoej1", title="Demostración R",
   ani.height = 400, ani.width = 600, interval = 1,
   title = "Punto fijo",
   description = c("Iteración de punto fijo", "Aproximación inicial inicial: x0 = 0.8, ", "
       Iteración xn_ = g(x_{n-1})" )     )
```

1.6 El Método de Newton

El método de Newton (llamado a veces método de Newton-Raphson), 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 supogamos 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

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Aplicando el mismo razonamiento a x_1 obtendríamos la aproximación $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$ y así sucesivamente.

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 1.5.1, 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, es decir este criterio de parada podría ser engañoso.

Método de Newton: Algoritmo e Implementación.

Figura 1.6: Método de Newton

Data: $f \in C^2[a,b]$, x_0, **tol** y **maxItr**.
Result: Si la iteración converge, una aproximación x_n de un cero de f en $[a,b]$ y una estimación del error.

$k = 0$;
$x_k = x_0$;
repeat
  $dx = \dfrac{f(x_k)}{f'(x_k)}$;
  $x_{k+1} = x_k - dx$;
  $x_k = x_{k+1}$;
  $k = k + 1$;
until $dx \leq$ **tol** or $k \leq$ **maxItr** ;
return x_k y dx

Implementación.

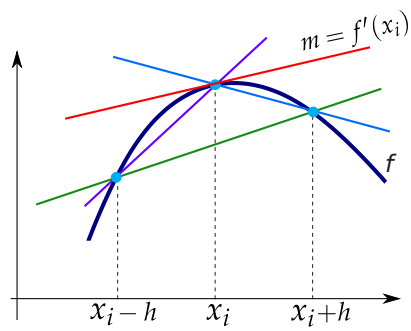
Una posible implementación recibiría a la función f y a su derivada como parámetros (por nombre). Una posible segunda implementación usaría la función **D()** de la base de R , para obtener la derivada. Ambas opciones son limitadas porque no siempre se puede obtener, de manera simbólica, la derivada de f. Para un programa de porpósito general es posible que la "derivada numérica" se ala mejor opción para implementar este función.

Geométricamente podemos deducir tres aproximaciones para $f'(x)$:

a.) Diferencia finita hacia adelante: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$

b.) Diferencia finita central: $f'(x) \approx \dfrac{f(x_i+h) - f(x-h)}{h}$

c.) Diferencia finita hacia atrás: $f'(x) \approx \dfrac{f(x) - f(x-h)}{h}$



Para la implementación vamos a usar "diferencia finita central". Usando polinomos de Taylor se puede establecer que

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} \quad \text{con error de orden} \quad O(h^2)$$

■ Código R 1.52: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 = abs(correccion)
    # 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  0.000000000000563  0.000000000068008
# -------------------------------------------------------------------------
```

```
# k=6  x=0.739085133215497  f(x)=5.62883073484954e-13
# Error estimado <=-6.80078666666667e-11
```

Ejemplo 1.11

Considere el vector `x = c(0.41040018, 0.91061564,-0.61106896, 0.39736684, 0.37997637)` y la función

$$g(\alpha) = \sum_{i=1}^{n} \frac{\mathbf{x}_i}{1+\alpha \mathbf{x}_i}.$$

Esta función tiene un único cero en $[-1,1]$. Podemos aproximar el cero con nuestra implementación.

Código R 1.53:Método de Newton con derivación numérica

```
x = c(0.91040018, -0.91061564,-0.61106896, -0.39736684, 0.37997637)
g = function(alpha)  sum( x/(1+alpha*x) )

# Ver gráfica
h = Vectorize(g)
curve(h,from =-1 , to = 1,  col = 2); abline(h=0, v=0, lty=3)

# Aproximar el cero
newtonDN(g, 0.5, 1e-10, 10)

# x_k                  f(x_k)              Error est.
# ---------------------------------------------------------------------------
#  0.001599885459682 -0.632409213961544  0.498400114540318
# -0.276265275549379  0.032414260035268  0.277865161009061
# -0.263289200045706 -0.000987590675503  0.012976075503673
# -0.263667728811509 -0.000019113938020  0.000378528765803
# -0.263674902271855 -0.000000757098332  0.000007173460346
# -0.263675223181762  0.000000064110994  0.000000320909907
# -0.263675199120908  0.000000002539194  0.000000024060854
# -0.263675198157917  0.000000000074898  0.000000000962991
# -0.263675198129210  0.000000000001436  0.000000000028707
# ---------------------------------------------------------------------------
#   k =  9  x =   -0.26367519812921  f(x) =  1.43596246005018e-12
#   Error estimado <=  -2.87073617021277e-11
```

Derivadas simbólica.
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).

```
D(expression(x*log(x)), "x")
#log(x) + x * (1/x)
```

Para poder usar una *expression,* se debe convertir en función con la función **eval().**

```
fx = D(expression(x*log(x)), "x")
#log(x) + x * (1/x)
fp = function(x) eval(fx)
# Calcular con fp
fp(3)
# [1] 2.098612
```

No hay una función base en R para calcular derivadas de orden superior. En vez de eso se puede usar una implementación (sugerida en la ayuda de **D()**) como la que sigue,

■ Código R 1.54: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))
```

También se puede usar el paquete **numDeriv** para el cálculo numérico eficiente de derivadas de primer y segundo orden.

1.7 Sistemas de ecuaciones lineales

Si $\mathbf{x} = (x_1, x_2, ..., x_n)^T$ y si $A_{n\times n}$ es invertible, sistema $A_{n\times n}X = B_{n\times 1}$ se puede resolver como $\mathbf{x} = A^{-1}B$. Sin embargo esta solución no es adecuada por la cantidad de error que se acarrea al calcular A^{-1} y hacer el producto.

En R se usa la función **solve()** para resolver sistemas de ecuaciones lineales $n \times n$.

```
A = matrix(c(4,    1,   0,  0,
             1,    5,   1,  0,
             0,    1,   6,  1,
             1,    0,   1,  4), nrow=4, byrow=TRUE)
B = c(1, 7, 16, 14)
det(A)
#[1] 420
solve(A, B)
#[1] 0 1 2 3
```

La función **solve** usa por defecto las subrutinas DGESV y ZGESV del paquete LAPACK. POr tanto, la función **solve()** usa por defecto descomposición LU con pivoteo parcial e intercambio de filas. En la página de netlib.org se lee,

> DGESV computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.
>
> The LU decomposition with partial pivoting and row interchanges is used to factor A as A = P * L * U, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

Descomposición LU.

Apliquemos eliminación gaussiana a la matriz a $A = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 5/2 \end{pmatrix}$.

Aplicamos la operación $\overline{F}_2 - \frac{1}{2}F_1$ 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 $\overline{F}_3 - \frac{1}{2}F_2$ para eliminar en la columna 2, fila 3,

$$\begin{matrix} \\ 1/2 \\ 0 \end{matrix} \begin{matrix} \\ \\ 1/2 \end{matrix} \begin{pmatrix} 4 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

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

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.

Pivoteo parcial en la descomposición LU. 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 $P^TA = LU$. Para obtner P, iniciamos con I y aplicamos a I las permutaciones de fila que aplicamos en A. Al final resolvemos $LY = P^Tb$ 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

$$AX = B \implies LUX = B \implies LUX = P^TB$$

por lo tanto, *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 $LY = P^Tb$ y luego resolvemos $UX = Y^*$.

Paquete Matrix.

La función **`lu()`** del paquete **`Matrix`** hace la descomposición LU aplicando pivoteo parcial. Esta función devuelve una lista **`(L,U,P)`.** Para acceder a los elemento de las lista usamos el operador **`$`.**

```
# 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
[2,]    3    1    3    1
[3,]    0    1    2    0
[4,]    3    2    4    1
```

Si no hubiera habido cambios de fila que tomar en consideración, la matriz P hubiera salido como

```
mlu$P
#4 x 4 sparse Matrix of class "pMatrix"
#[1,] | . . .
#[2,] . | . .
#[3,] . . | .
#[4,] . . . |
```

Por lo que hubo cambios de fila.

Ejemplo 1.12

Use la función **lu()** del paquete **Matrix** para resolver (usando descomposición LU con pivoteo parcial), los sistemas $AX = B$, $AX = B'$ y $AX = B''$ donde

$$A = \begin{pmatrix} 4 & 0 & 1 & 1 \\ 3 & 1 & 3 & 1 \\ 0 & 1 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{pmatrix}, \qquad B = (5,6,13,1)^T, \qquad B' = (6,7,8,9)^T \quad \text{y} \quad B'' = (1,2,5,2)^T$$

Solución:

```
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)
B  = c(5,6,13,1)
Bp = c(6,7,8,9)
Bpp = c(1,2,5,2)

mlu = lu(A)
mlu = expand(mlu)
LL  = mlu$L
UU  = mlu$U
Pt  = t(mlu$P)
# Solución de los sistemas
Y = solve(LL, Pt%*%B) # aplicar cambios de fila a B
solve(UU, Y)
#Solución:
#     [,1]
#[1,]   12
#[2,]  -23
#[3,]   18
#[4,]  -61
# El mismo resultado se obtiene aplicando
solve(A,B)
#[1]  12 -23  18 -61
# Los otros casos-----------------------------------
Y = solve(LL, Pt%*%Bp) # aplicar cambios de fila a Bp
solve(UU, Y)

Y = solve(LL, Pt%*%Bpp) # aplicar cambios de fila a Bpp
```

```
solve(UU, Y)
```

1.7.1 Sistemas $m \times n$.

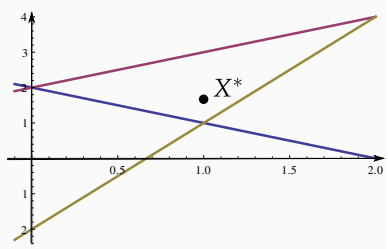
Consideremos el sistema $A_{m\times n}X = B$ con $m > n$. En general, para un B arbitrario, lo mejor que podemos hacer es encontrar un vector X^* que *minimice* la norma euclidiana:

$$X^* = \text{mín}_{X\in\mathbb{R}^n}||AX - B||_2^2$$

Al vector X^* le llamamos la "solución por mínimos cuadrados".

Ejemplo 1.13

La ecuación cartesiana de uan recta es $ax + by = c$. Consideremos las tres rectas de ecuación $L_1 : x + y = 2$, $L_2 : x - y = -2$ y $L_3 : 3x - y = 2$. 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 mín(m,n). En este caso, la matriz A^TA es simétrica y definida positiva y además la solución X^* existe y es única. De hecho, X^* es la solución del sistema

$$A^TAX = A^TB$$

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: $\begin{cases} 1+0.9666667 &= 1.966667 \\ 1-0.9666667 &= 0.0333333 \\ 3\cdot 1-0.9666667 &= 2.033333 \end{cases}$

1.8 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 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.

1.8.1 Métodos iterativos clásicos.

Para resolver numéricamente un sistema $Ax = B$ con A no-singular, los métodos iterativos inician con una aproximación $\boldsymbol{x}_0$ de la solución $\boldsymbol{x}^*$ y calculan una sucesión $\boldsymbol{x}_k \to \boldsymbol{x}^*$. En general tales iteraciones se pueden escribir como $\boldsymbol{x}_{k+1} = g_k(\boldsymbol{x}_k)$ donde g_k es una sucesión de funciones. Si $g_k = g$ el método se dice "estacionario".

La $i-$esima ecuación del sistema $Ax = B$ se puede escribir como,

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n = b_n$$

Resolviendo para x_i obtenemos,

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^{n} a_{ij}x_j\right)$$

Si $\boldsymbol{x}_k = (\boldsymbol{x}_k(1), \boldsymbol{x}_k(2), ..., \boldsymbol{x}_k(n))$ es una aproxi-mación de $\boldsymbol{x}^*$ entonces podemos calcular una nueva aproximación $\boldsymbol{x}_{k+1}$ como

$$\boldsymbol{x}_{k+1}(i) = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\boldsymbol{x}_k(j) - \sum_{j=i+1}^{n} a_{ij}\boldsymbol{x}_k(j)\right)$$

Ahora observe que $A = L + D + U$ donde L y U son las matrices triangular inferior y triangular superior de A respectivamente, y D es la diagonal de A, es decir,

$$A = \begin{pmatrix} 0 & & & \\ a_{21} & 0 & & \\ \vdots & \ddots & \ddots & \\ a_{n1} & \dots & a_{n,n-1} & 0 \end{pmatrix} + \begin{pmatrix} a_{11} & & & \\ & a_{2,2} & & \\ \vdots & \ddots & \ddots & \\ & & & a_{n,n} \end{pmatrix} + \begin{pmatrix} 0 & a_{12} & \dots & a_{1n} \\ & \ddots & \ddots & \vdots \\ & & 0 & a_{n-1,n} \\ & & & 0 \end{pmatrix}$$

Entonces si D es invertible,

$$\begin{aligned} \boldsymbol{x}_{k+1}(i) &= \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\boldsymbol{x}_k(j) - \sum_{j=i+1}^{n} a_{ij}\boldsymbol{x}_k(j)\right) \\ &= \frac{1}{a_{ii}}\left(b_i - (\boldsymbol{L}\boldsymbol{x}_k)_i + (\boldsymbol{U}\boldsymbol{x}_k)_i\right) \end{aligned}$$

Y en general, como $-A + D = -(L + U)$, se tiene

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + D^{-1}(B - \boldsymbol{L}\boldsymbol{x}_k - \boldsymbol{U}\boldsymbol{x}_k) = \boldsymbol{x}_k + D^{1}(B - \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{D}\boldsymbol{x}_k)$$

Esta es la iteración de Jacobi:

(Jacobi)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + D^{-1}\boldsymbol{r}_k \quad \text{con} \quad \boldsymbol{r}_k = B - \boldsymbol{A}\boldsymbol{x}_k$$

El método SOR (Successive Over Relaxation method) se obtiene introduciendo un parámetro de aceleración $\omega \in]1,2[$

(SOR)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \left(\boldsymbol{L} + \omega^{-1} D^{-1}\right) \boldsymbol{r_k} \quad \text{con} \quad \boldsymbol{r_k} = B - \boldsymbol{A}\boldsymbol{x}_k$$

El método de Gauss-Seidel se obtiene con $\omega = 1$ en el método SOR.

(Gauss-Seidel)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \left(\boldsymbol{L} + D^{-1}\right) \boldsymbol{r_k} \quad \text{con} \quad \boldsymbol{r_k} = B - \boldsymbol{A}\boldsymbol{x}_k$$

Implementación.

La versión matricial de estos métodos hacen muy directa la implementación en R .

De los paquetes se puede aprender mucho. La implementación del paquete **pracma** es

■ **Código R 1.55:Jacobi y Gauss-Seidel — Paquete pracma**

```
eye <- function(n, m = n) {
    stopifnot(is.numeric(n), length(n) == 1,
              is.numeric(m), length(m) == 1)
    n <- floor(n)
    m <- floor(m)
    if (n <= 0 || m <= 0) return(matrix(NA, 0, 0))
    else                  return(base::diag(1, n, m))
}

tril <- function(M, k = 0) {
    if (k == 0) {
        M[upper.tri(M, diag = FALSE)] <- 0
    } else {
        M[col(M) >= row(M) + k + 1] <- 0
    }
    return(M)
}

#--
itersolve <- function(A, b, x0 = NULL,
                      nmax = 1000, tol = .Machine$double.eps^(0.5),
```

```
                    method = c("Gauss-Seidel", "Jacobi", "Richardson")) {
stopifnot(is.numeric(A), is.numeric(b))

n <- nrow(A)
if (ncol(A) != n)
    stop("Argument 'A' must be a square, positive definite matrix.")
b <- c(b)
if (length(b) != n)
    stop("Argument 'b' must have the length 'n = ncol(A) = nrow(A).")
if (is.null(x0)) {
    x0 <- rep(0, n)
} else {
    stopifnot(is.numeric(x0))
    x0 <- c(x0)
    if (length(x0) != n)
        stop("Argument 'x0' must have the length 'n=ncol(A)=nrow(A).")
}

method <- match.arg(method)

if (method == "Jacobi") {
    L <- diag(diag(A))
    U <- eye(n)
    beta <- 1; alpha <- 1
} else if (method == "Gauss-Seidel") {
    L <- tril(A)
    U <- eye(n)
    beta <- 1; alpha <- 1
} else {  # method = "Richardson"
    L <- eye(n)
    U <- L
    beta <- 0
}

b <- as.matrix(b)
x <- x0 <- as.matrix(x0)
r <- b - A %*% x0
r0 <- err <- norm(r, "f")

iter <- 0
while (err > tol && iter < nmax) {
    iter <- iter + 1
    z <- qr.solve(L, r)
    z <- qr.solve(U, z)
    if (beta == 0) alpha <- drop(t(z) %*% r/(t(z) %*% A %*% z))
    x <- x + alpha * z
    r <- b - A %*% x
```

```
        err <- norm(r, "f") / r0
    }

    return(list(x = c(x), iter = iter, method = method))
}

##--- Pruebas
A = matrix(c(4, -1, -1, 0, 0, 0,
             -1, 4, 0, -1, 0, 0,
             -1, 0, 4, -1, -1, 0,
              0, -1, -1, 4, 0, -1,
              0, 0, -1, 0, 4, -1,
              0, 0, 0, -1, -1, 4), nrow=6, byrow=TRUE)

b = c(1, 5, 0, 3,1,5)
x0 = c(0.25, 1.25, 0, 0.75, 0.25, 1.25)
itersolve(A, b, tol = 1e-8, method = "Gauss-Seidel")

#$x
#[1] 1 2 1 2 1 2

#$iter
#[1] 19

#$method
#[1] "Gauss-Seidel"
```

1.9 Sistemas de ecuaciones no lineales.

En R hay varios paquetes para resolver sistema no lineales del tipo $F(x) = 0$ donde $F : \mathbb{R}^p \to \mathbb{R}^p$ es una función nolineal con derivadas parciales continuas. Por ejemplo **multiroot, rootsolve, nleqslv** y **BB**

En la descripción de la función **BBsolve()** del paquete **BB** se indica

> '...the R package BB is a (slightly) modified version of Varadhan and Gilbert (2009), published in the Journal of Statistical Software. The function BBsolve in BB can be used for this purpose. We demonstrate the utility of these functions for solving: (a) large systems of nonlinear equations, (b) smooth, nonlinear estimating equations in statistical modeling, and (c) non-smooth estimating equations arising in rank-based regression modeling of censored failure time data. The function BBoptim can be used to solve smooth, box-constrained optimization problems. A main strength of BB is that, due to its low memory and storage requirements, it is ideally suited for solving high-dimensional problems with thousands of variables.'

Por ejemplo, consideremos el problema de determinar numéricamente la intersección entre la circunferencia $x^2 + y^2 = 1$ y la recta $y = x$. Usamos una aproximación inicial $(1,1)$.

```
require(BB)
sistema = function(p){
 f    = rep(NA, length(v))
 f[1] = p[1]^2+p[2]^2-1  # = 0
 f[2] = p[1]-p[2]        # = 0
 f
}
p0 = c(1,1)
BBsolve(par=v0, fn=sistema)
#  Successful convergence.
#  sol$par
#[1] 0.7071068 0.7071068
```

En la documentación del paquete nos encontramos con un ejemplo con 10000 ecuaciones.

```
trigexp = function(x) {
 n = length(x)
 F = rep(NA, n)
 F[1] = 3*x[1]^2 + 2*x[2] - 5 + sin(x[1] - x[2]) * sin(x[1] + x[2])
 tn1 = 2:(n-1)
 F[tn1] = -x[tn1-1] * exp(x[tn1-1] - x[tn1]) + x[tn1] *
 ( 4 + 3*x[tn1]^2) + 2 * x[tn1 + 1] + sin(x[tn1] -
 x[tn1 + 1]) * sin(x[tn1] + x[tn1 + 1]) - 8
 F[n] = -x[n-1] * exp(x[n-1] - x[n]) + 4*x[n] - 3
 F
 }
 n = 10000
 p0 = runif(n) # n initial random starting guesses
 sol = BBsolve(par=p0, fn=trigexp)
 sol$par
 # 10 mil soluciones
 # [1] 1 1 1 1 1 ...
```

1.10 Interpolación

Dados $n+1$ *pares* $(x_0,y_0),(x_1,y_1),...,(x_n,y_n)$, siendo todos los x_i's distintos, y $y_i = f(x_i)$ para alguna función f; el polinomio interpolante para estos datos es el único polinomio $P_n(x)$ de grado $\leq n$ tal que

$$P_n(x_i) = y_i, \ \ i = 0,1,2,...,n$$

La forma de Lagrange del polinomio interpolante es $P_n(x)$ de grado $\leq n$ que pasa por los $n+1$ puntos $(x_0,y_0),(x_1,y_1),...,(x_n,y_n)$ (con $x_i \neq x_j$ si $i \neq j$) es

$$P_n(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + ... + y_n L_{n,n}(x)$$

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

Implementación en R.

Para hacer una implementación vectorizada, recordemos que en el ejemplo 1.5 habíamos visto que si `x = c(x0,x1,...,xn)` y si

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

entonces si $n = 4$, si hacemos `mN = a - X` y `diag(mN)= 1,` obtenemos

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

y si hacemos `mD = X - t(X)` y `dia(mD)=1,` obtenemos

$$\texttt{mD} = \begin{pmatrix} 1 & x_1-x_0 & x_2-x_0 & x_3-x_0 \\ x_0-x_1 & 1 & x_2-x_1 & x_3-x_1 \\ x_0-x_2 & x_1-x_2 & 1 & x_3-x_2 \\ x_0-x_3 & x_1-x_3 & x_2-x_3 & 1 \end{pmatrix}$$

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

```
Lnk(a)= prod(N[k, ])/prod(D[ ,k])
```

Código R 1.56:Forma de Lagrange del polinomio interpolante

```
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)
}
```

```
# --- 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 1.57: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
```

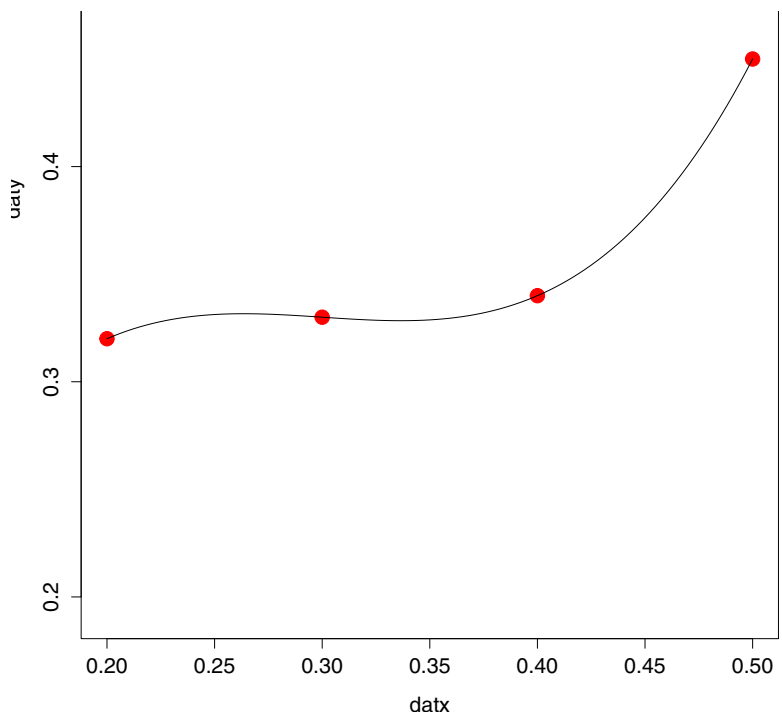


Figura 1.7: Ajuste de los datos con un polinomio interpolante

Oscilación. Como mencionabamos antes, los polinomios interpolantes de grado alto "oscilan" y Por ejemplo,

■ **Código R 1.58:"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)
```

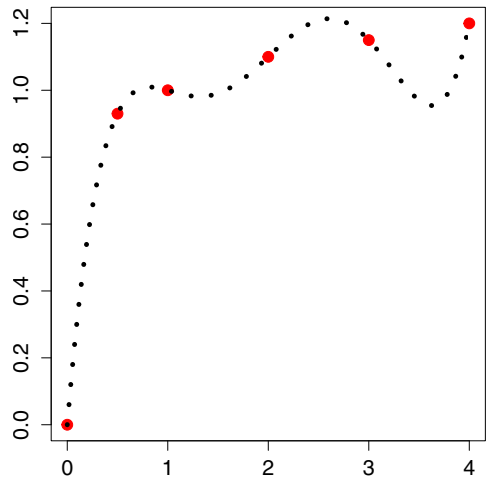


Figura 1.8: Ajuste de los datos con un polinomio interpolante de grado 5

1.11 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. La forma modificada y la forma baricéntrica de Lagrange son útiles cuando queremos interpolar una función en todo un intervalo con un con un polinomio interpolante.

Supongamos que tenemos $n+1$ nodos distintos $x_0, x_1, ..., x_n$. 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

$$\omega_k = \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{1}{x_k - x_i}, \quad k = 0, 1, ..., n.$$

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 1.1

La *forma modificada* del polinomio de Lagrange se escribe como

$$P_n(x) = \ell(x) \sum_{j=0}^{n} \frac{\omega_j}{x - x_j} y_j \tag{1.2}$$

Definición 1.2

La *forma baricéntrica* del polinomio de Lagrange se escribe

$$P_n(x) \begin{cases} = & y_i & \text{si} \quad x = x_i, \\ = & \dfrac{\displaystyle\sum_{k=0}^{n} \frac{\omega_k}{x - x_k} y_k}{\displaystyle\sum_{k=0}^{n} \frac{\omega_k}{x - x_k}}, & \text{si} \quad x \neq x_i \end{cases} \tag{1.3}$$

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
```

1.12 Integración

integrate() es la función en la base de R para integrar. Usa la librería QUADPACK de Netlib. Por ejemplo, podemos aproximar $\int_0^{\pi} e^{-x} \cos x \, dx$ con el código

```
f = function(x) exp(-x) * cos(x)
integrate(f, 0, pi)
# 0.521607 with absolute error < 7.6e-15
```

integrate() retorna una lista. La aproximación numérica se puede obtener con **q$value**

```
f = function(x) exp(-x) * cos(x)
q = integrate(f, 0, pi)
q$value
# [1] 0.521607
```

La ayuda recomienda "vectorizar las funciones"

```
f = function(x) max(0,x)
q = integrate(Vectorize(f), -1, 1)
q$value
# [1] 0.5
```

Aunque de más cuidado, se puede calcular integrales impropias, el análsis del resultado queda, por supuesto, bajo la responsabilidad del usuario.

```
f = function(x) sqrt(x) * exp(-x)
integrate(f, 0, Inf)
# 0.8862265 with absolute error < 2.5e-06
```

Ejemplo 1.14 (Función de Bessel $J_0(z)$)

Consideremos una de las funciones de Bessel, definida por una integral,

$$J_0(z) = \int_0^{\pi} \cos(z\cos t)\, dt$$

Como es una función de dos variables, necesitamos un pequeño truco para definirla en R . Para poder evaluar en un vector se requiere "vectorizar", en este caso es suficiente aplicar la función **sapply()** que devuelve, en este caso, un vector. Hecho esto podemos entonces hacer la representación gráfica de $J_0(z)$.

■ **Código R 1.59:Función de Bessel** $J_0(z)$

```
 J0zt = function(z){
           J0t = function(t){ cos(z*cos(t)) }
           return(J0t)
                }
J0 = function(z) integrate(J0zt(z), 0, pi)$value
# Vectorización.
# La función sapply(), en este caso,  un vector
J0v = function(z) sapply(z, J0)
# Gráfica (ya J0v está vectorizada)
curve(J0v, 0,41, n= 200, lwd=3)
abline(h=0, v=0, lty=3)
```

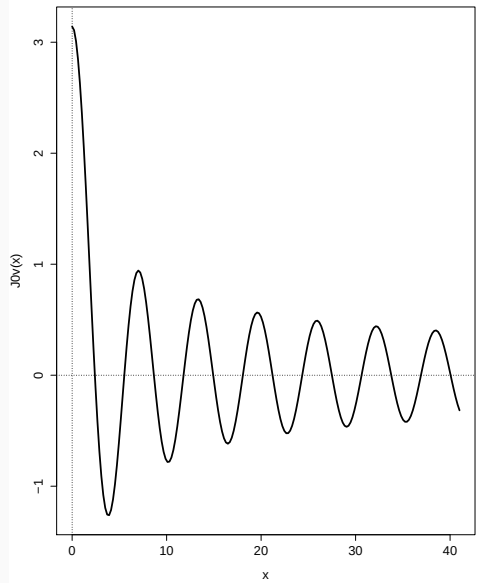


Figura 1.9: Función de Bessel $J_0(z)$

Ahora podemos usar el paquete **gridExtra** para imprimir una tabla bonita de valores

■ **Código R 1.60:Tabla de valores para** $J_0(z)$

```
# Cálculo de algunos valores
dat = seq(0.1, 1, by = 0.1)
n   =  length(dat)
x   = matrix(J0v(dat),n, byrow=T)
x   = cbind(dat,x)
dimnames(x) = list(NULL, head(c("x","J0(x)"), ncol(x))
    )
library(gridExtra)
#pdf("data_output.pdf", height=11, width=8.5)
grid.table(x)
dev.off()
```

x	J0(x)
0.1	3.13374357933101
0.2	3.11025517965829
0.3	3.07130343408127
0.4	3.01718001339435
0.5	2.94828985204357
0.6	2.865147779201
0.7	2.76837423798582
0.8	2.65869013022686
0.9	2.53691083178071
1	2.40393943063441

Figura 1.10: $J_0(z)$ con $z \in [0.1,]$

Finalmente podemos eproximar numericamente sus *ceros* en el intervalo $[0,41]$ con el paquete **rootSolve**

■ **Código R 1.61:Ceros de** $J_0(z)$ **en** $[0,41]$

```
require(rootSolve)
uniroot.all(J0v, c(0,41) )
#  [1]  2.404815  5.520063  8.653732 11.791533 14.930919 18.071065 21.211638
    24.352470 27.493479
#  [10] 30.634608 33.775818 36.917098 40.058428
```

Cuadratura gaussiana. El paquete **gaussquad** es una colección de funciones para hacer cuadratura gaussiana. Por ejemplo, la función **legendre.quadrature()** ejecuta cuadratura gaussiana con nodos de Legendre en $[-1,1]$.

Veamos un ejemplo con una integral difícil: $\int_{-1}^{1} \text{sen}(x+\cos(10e^x)/3)\,dx$

```
library(gaussquad)
f = function(x) sin(x+cos(10*exp(x))/3)
# Con solo 4 nodos: un resultado no muy bueno
Lq = legendre.quadrature.rules(4)[[4]] #Nodos y pesos
# xi = Lq$x; wi = Lq$w
legendre.quadrature(f, Lq, lower = -1, upper = 1)
#[1] 0.1350003
# Comparación
integrate(f, -1,1)
# 0.03250365 with absolute error < 6.7e-07
# Mejor resultado con 51 nodos
Lq = legendre.quadrature.rules(51)[[51]] #Nodos y pesos
legendre.quadrature(f, Lq, lower = -1, upper = 1)
# [1] 0.03250365
# Comparación
integrate(f, -1,1)
# [1] 0.03250365
```

Paquete pracma. En el paquete **pracma** hay varias rutinas de integración. La función **quad()** usa la regla adaptiva de Simpson mientras que **quad1()** usa cuadratura de Lobatto (similar a cuadratura gaussiana, excepto que incluye los extremos del intervalo), La función **cotes()** usa las fórmulas de Newton-Cotes de grados 2 hasta 8. El método de Romberg está implementado en **romberg()** y la función **quadcc()** implementa la cuadratura de Clenshaw-Curtis (la rival de Gauss-Kronrod)

```
f = function(x) sin(x+cos(10*exp(x))/3)
romberg(f, -1,1, tol=1e-10)
# $value
#[1] 0.03250365

#$iter
#[1] 9

#$rel.error
#[1] 1e-10

 quadcc(f, -1,1)
# [1] 0.03250365
```

Integral doble. Hay dos paquetes **cubature** y **R2cuba** para hacer integración multidimensional. En el paquete **cubature** se puede usar la función **adaptIntegrate().** El paquete **pracma** tiene las funciones **integral2()** e **integra3()** para aproximar integrales dobles y triples, respectivamente.

Por ejemplo, consideremos la integral $\int_0^1 \int_0^1 \frac{1}{1+x^2+y^2}\, dx$

```
require(cubature)
f = function(x) 1 / (1 + x[1]^2 + x[2]^2)
(q2 = adaptIntegrate(f, c(0, 0), c(1, 1)))

#$integral
#[1] 0.6395104

#$error
#[1] 4.5902e-06

#$functionEvaluations
#[1] 119

#$returnCode
#[1] 0
```

Para dimensiones ≥ 4 se usa el método Monte Carlo. En el paquete **R2cuba** viene la función $vegas()$. También se puede usar el paquete **SparseGrid.**

1.12.1 Implementaciones en R.

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) $x_0, x_1, ..., x_n$ 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

$$\int_a^b f(x)\, dx = \int_a^b P_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)(x-x_1)\cdots(x-x_n)\, dx \tag{1.4}$$

1.13 Regla del Simpson.

En la regla de Simpson se usa interpolación cuadrática.

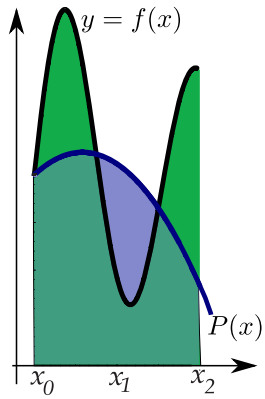


Figura 1.11

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_a^b P_2(x) + (x-x_0)(x-x_1)(x-x_2)f^{(4)}(\xi(x))/2\,dx \\ &= \frac{h}{3}[f(a)+4f(x_1)+f(b)] + \int_a^b (x-x_0)(x-x_1)(x-x_2)f^{(4)}(\xi(x))/2\,dx\end{aligned}$$

(Regla compuesta de Simpson).

Si n es par,

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_2} f(x)\,dx + \int_{x_2}^{x_4} f(x)\,dx + \ldots + \int_{x_{n-2}}^{x_n} f(x)\,dx \\ &= \frac{h}{3}\left[f(a)+f(b)+4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i})\right] - \frac{1}{180}(b-a)h^4 f^{(4)}(\xi)\end{aligned}$$

con $\xi \in]a,b[$, $h = \dfrac{b-a}{n}$ y $x_i = a + i\cdot h$, $i = 0,1,2,\ldots,n$.

La simplificación del término de error se hace como se hizo en la regla del Trapecio.

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.

Algorithm 1.2: Regla adaptativa de Simpson

Data: $f(x)$, a, b y $n \in \mathbb{N}$ par.

Result: Aproximación de $\int_a^b f(x)\,dx$.

suma$= 0$;
$h = (b-a)/n$;
for $i = 0$ **to** $(n-1)$ **do**
    suma=suma$+f[a+i\cdot h]+4\cdot f[a+(i+1)\cdot h]+f[a+(i+2)\cdot h]$;
    $i = i+2$;
return suma$\cdot h/3.0$

En R podemos hacer una implementación vectorizada,

■ Código R 1.62:Regla de Simpson

```
simpson2 = 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
  y =  fun(a+(0:n)*h)   # f(a), f(a+h),...,f(a+i*h),...
  h/3 * ( fun(a) + fun(b) + 4*sum(y[i1]) + 2*sum(sum(y[i2]) ) )
}

#--- Pruebas
f = function(x) sin(x)/x
simpson2(f, 1, 2, 1000)
#[1] 0.6597164
integrate(f, 1,2)$value
[1] 0.6593299
```

Discretización. Las reglas del trapecio y Simpson son adecuadas cuando solo tenemos una tabla de datos de la función. En este caso, una implementación de la regla de Simpson solo requiere los nodos x_i y los valores $y_0, y_1, ..., y_n$ (un número umpar de datos!),

```
simpsonD = function(y, h) {
n = length(y)
if (n%%2 != 1) stop("Simpson requiere lenght(y) impar.")
```

```
 i1 = seq(2, n-1, by=2)
 i2 = seq(3, n-2, by=2)
 h/3 * (y[1] + y[n] + 4*sum(y[i1]) + 2*sum(y[i2]))
}

## --- Pruebas
yi = c(0.3, 0.5, 0.05, 0.4)
simpson(yi, 0.1)
[1] 0.521607
```

1.14 Ecuaciones diferenciales

Las ecuaciones diferenciales (EDOs) están presentes en todo lugar en ciencias e ingienería. R tiene varios paquetes para resolver numéricamente EDOs. Una visión general de los paquetes que se pueden usar se puede obtner 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 equa- tions. The functions provide an interface to the FORTRAN functions lsoda, lsodar, lsode, lsodes of the ODEPACK collection, to the FORTRAN func- tions 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 1.15 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 la transferencia de calor entre el cuerpo y el entorno externo puede ser descrita con la ley de

Stefan-Boltzmann,

$$v(t) = \epsilon\gamma S\left(T^4(t) - T_e^4\right)$$

donde t es tiempo y ϵ es la constante de Boltzmann ($\epsilon = 5.6 \times 10^{-8} J/\mathrm{m}^2K^2\mathrm{s}$, 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 v 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 v. En consecuencia, si $T(0) = T_0$, el cálculo de $T(t)$ requiere la solución de la ecuación diferencial ordinaria

$$\frac{dT}{dt} = -\frac{v(t)}{mC} \qquad (*)$$

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 1m y masa igual a 1Kg. Asuma que $T_0 = 180\,\mathrm{K}$, $T_e = 200K$, $\gamma = 0.5$ y $C = 100J/(Kg/K)$. Hacer una representación gráfica del resultado.

Solución:
$S = 6\mathrm{m}^2$. Sustituyendo los valores dados, el problema queda

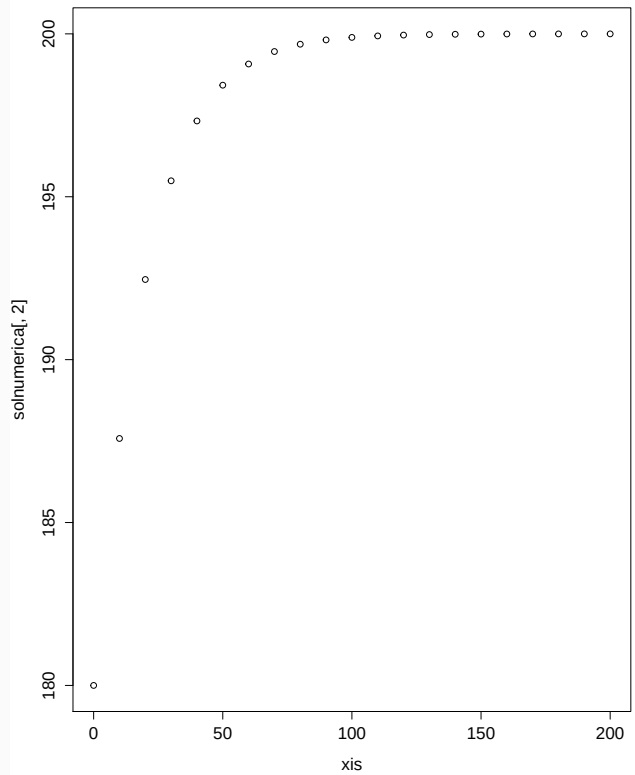
$$\frac{dT}{dt} = -1.68 * 10^{-9} * T(t)^4 + 2.6880 \quad \text{con} \quad T(0) = 180 \quad \text{y} \quad t \in [0,200]$$

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(xis, solnumerica[,2] )
```

```
colnames(tabla) = c("ti", "    Ti     ")
tabla
# Representación
plot(xis, 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 1.16

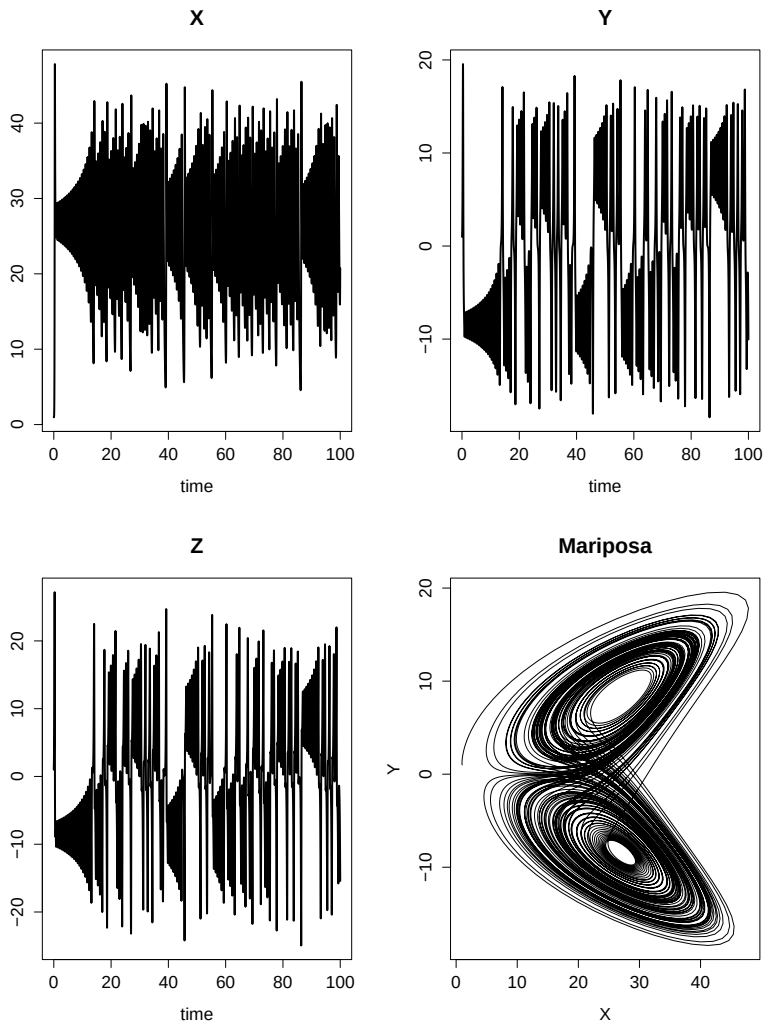
Las ecuaciones de Lorentz es un sistema dinámico con comportamiento caótico (el primero en ser descrito). ESte modelo consiste de tres ecuaciones diferenciales que expresan la dinámica de tres variables x,y,z que se asume, representan un comportamiento idealizado de la atmósfera de la tieraa. El modelo 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")
```



1.14.1 El método de Euler

Como un ejemplo de implementación en R podemos usar el método de Euler (1758): 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,

$$y(t_{i+1}) \approx y_{i+1} = y_i + h f(t_i, y_i), \quad i = 0, 1, ..., n$$

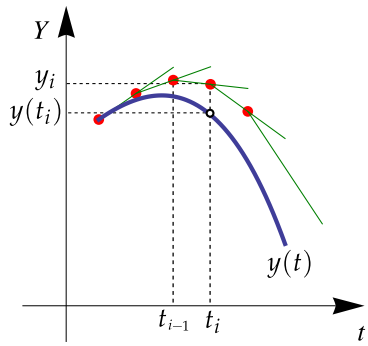


Figura 1.12: $y(t_i) \approx y_i = y_{i-1} + hf(t_{i-1}, y_i)$.

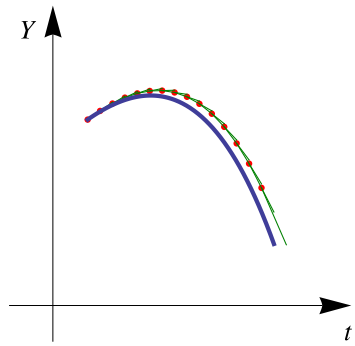


Figura 1.13: Tangentes en (t_i, y_i), $i = 0, 1, ...15$.

Implementación. En la implementación que sigue, se usa como problema de prueba la ecuación diferencial del ejemplo 1.15

Código R 1.63:Método de Euler

```
euler1 = function(init, xis ,func) {
 n = length(xis)
 h = xis[2] - xis[1]
 v.num = vector(length = n)
 v.num[1] = init
 for (j in 1:(n-1)) {
   v.num [j+1] = v.num [j] +
     h*func(xis[j], v.num[j]) }
 v.num}
 # --- Pruebas
f = function(t,y) -1.68*10^(-9)*y^4+2.6880
xis= seq(0,200,200/20)
euler1(180, xis, f)
# [1] 180.0000 189.2440 194.5765 197.3757 198.7590 199.4200 199.7304 199.8751
# [9] 199.9422 199.9732 199.9876 199.9943 199.9974 199.9988 199.9994 199.9997
# [17] 199.9999 199.9999 200.0000 200.0000 200.0000
```

1.15 Conclusión

Hemos hecho una pequeña excursión con una parte pequeña del lenguaje R , observando como se puede aplicar en temas usuales de métodos números. Se observa que se mantiene la simplicidad y la efectividad a la hora de implementar algoritmos y hacer experimentos, además de poder usar paquetes con las implementaciones de los algoritmos más avanzados para comparar o para resolver problemas aplicados reales.

Bibliografía

[1] Owen Jones, Robert Maillardet, and Andrew Robinson. *Introduction to Scientific Programming and Simulation using R.* CRC Press. 2009.

[2] Alfio Quarteroni, Fausto Saleri. *Scientific Computing with MATLAB and Octave.* Springer, 2003.

[3] Norman Matloff. *The art of R programming.* No Starch Press, Inc. 2011

[4] Victor A. Bloomfield. *Using R for Numerical Analysis in Science and Engineering.* CRC Press. 2014.

[5] Arthur Charpentier, ed. *Computational Actuarial Science with R.* CRC Press. 2015.