Ejercicio 1

El método de Newton es altamente eficiente en funciones convexas, pero su comportamiento puede ser impredecible en funciones no convexas. Este ejercicio explora la sensibilidad del método al punto inicial y la naturaleza de los puntos críticos que encuentra. Consideremos la función “Three-Hump Camel”, una prueba estándar para algoritmos de optimización, definida como:

$f(x,y) = 2x2 - 1.05x4 + x6\ 6 + xy + y2$

1. **Derivación analítica:**

    a. Calcule el vector gradiente $\nabla f(x,y)$ y la matriz Hessiana $Hf(x,y)$ de la función.

2. **Implementación:**

    a. implemente el método de Newton multivariado para encontrar los mínimos de esta función. El código debe permitir especificar un punto de partida inicial.

3. **Experimentación:**

    a. Ejecute su algoritmo desde los siguientes tres puntos iniciales:

        i. x0 =(1.5,1.0)
        ii. x0 =(−1.5,−1.0)
        iii. x0 =(0.1,−0.2), cerca de un máximo local

4. Para cada caso, informe el punto final al que converge el algoritmo, el valor de la función en ese punto y el número de iteraciones requeridas.

5. Genere un gráfico de contorno de la función y trace la trayectoria de optimización (los puntos xk en cada iteración) para cada uno de los tres puntos iniciales.

6. Discuta por qué diferentes puntos de partida conducen a diferentes resultados y analice el comportamiento del algoritmo cuando se inicia cerca de un punto no mínimo.

Soluciones:

Función de manera de Código de lenguaje R

```r
f_objetivo <- function(x,y){
  return (2*(x**2) - 1.05*(x**4) + ((x**6)/6) + x*y + y**2 )
}
#-----------------------------------------------------------------

# definir parametros
f <- function(param){
  x <- param[1]
  y <- param[2]

  return(f_objetivo(x,y))
}
#
```

Gradiente de la función y se matriz Hessiana mediante código de lenguaje R

```r
# definimos  el tirnagulo F
gran_f <- function(param){
  x <- param[1]
  y <- param[2]

  dx <- 4*x - (1.05*4*(x**3)) + (x**5) + y
  dy <- x + 2*y

  return(c(dx,dy))
}
#-----------------------------------------------------------

# Definimos la matriz hessianade la función
hess_f<-function(param){
  x <-param[1]
  y <-param[2]

  # Segundas derivadas
  d2x <- 4 - 1.05*12*(x**2) + 5*(x**4)
  d2y <- 2
  dxdy <- 1

  return(matrix(c(d2x, dxdy,
                  dxdy, d2y),
                ncol = 2, byrow = TRUE))
}

#-----------------------------------------------------------
```

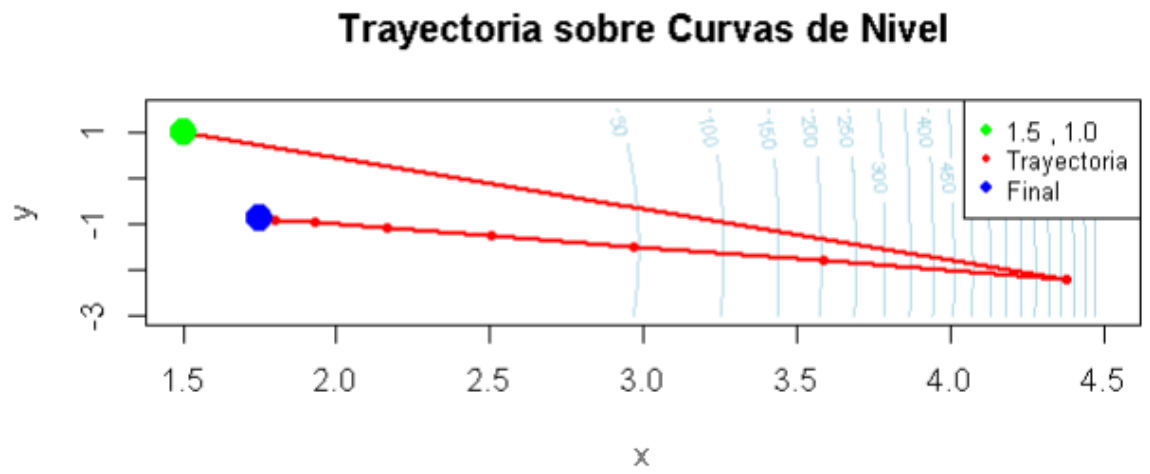
Cómo obtener el nuevo X mediante código en lenguaje R

```r
for (i in 1:max_iter) {

  #guardamos los parametros "nuevos"
  historial_x[i] <- x_actual[1]
  historial_y[i] <- x_actual[2]
  historial_f[i] <- f(x_actual)
  #-----------------------------------------------------------------

  # X - ( hessiano **-1  * gradiente f )
  x_nuevo<-x_actual - solve(hess_f(x_actual)) %*% gran_f(x_actual)
  #-----------------------------------------------------------------

  if (norm(gran_f(x_nuevo), "2")< tol){
    cat("Convergió en iteración:", i, "\n")
    break
  }
  #-----------------------------------------------------------------

  x_actual<-x_nuevo
}
```

X inicial en el punto (1.5 , 1)

```
> print(resultados)
$punto_minimo
[1]  1.7475524 -0.8737762

$valor_minimo_Fx
[1] 0.2986384

$iteraciones
[1] 10

$gradiente
[1] 4.292686e-07

>
```

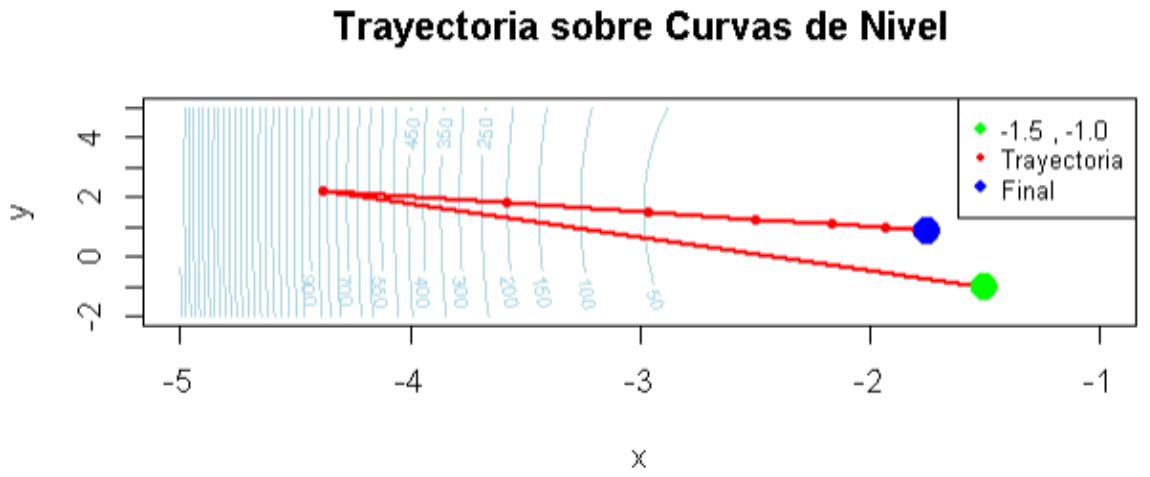
Gráfico



X inicial en punto (-1.5 , -1)

```
$punto_minimo
[1] -1.7475524  0.8737762

$valor_minimo_Fx
[1] 0.2986384

$iteraciones
[1] 10

$gradiente
[1] 4.292686e-07

>
```

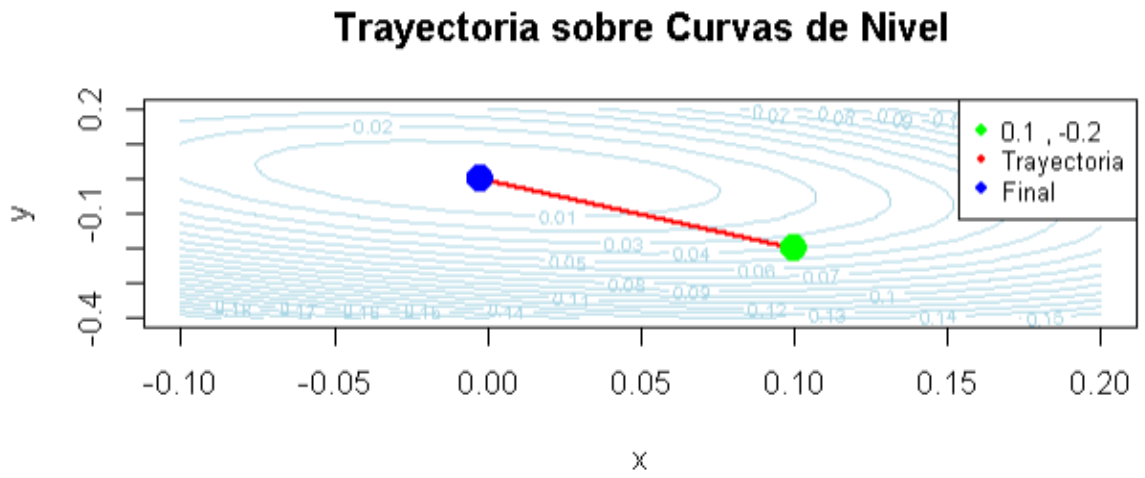
Gráfico



X inicial en el punto (0.1 , -0.2)

```
$punto_minimo
[1]  3.649305e-08 -1.824652e-08

$valor_minimo_Fx
[1] 2.330549e-15

$iteraciones
[1] 2

$gradiente
[1] 1.277257e-07

>
```

Gráfico



La razón de porqué distintos inicios generan distintos puntos mínimos es porque cada uno es mínimo pero puede ser unos locales y el otro global, y esta caída en un mínimo local y no en uno global se debe que esta función solo se acerca el mínimo más cercano de su punto inicial por eso también si elegimos un punto cerca del mini global se obtiene aquel mínimo, pero si elegimos un punto inicial lejano al mínimo global lo más probable es que llegue a un mínimo local. También hay que mencionar que entre más lejano esté el punto inicial de un mínimo (ya sea local o global) más iteraciones le tomará llegar

Ejercicio 2

En ingeniería, es común necesitar ajustar un modelo teórico a datos experimentales. Este ejercicio consiste en usar el método de Newton para encontrar los parámetros de un modelo de oscilación 1 amortiguada, un problema fundamental en el procesamiento de señales y el control de sistemas. Se ha medido una señal que sigue el modelo:

$$y(t) = Ae^{-\lambda t}\cos(\omega t + \phi)$$

El objetivo es encontrar el vector de parámetros $\beta = [A, \lambda, \omega, \phi]$ que mejor se ajuste a un conjunto de mediciones (ti, yi). Para ello, se debe minimizar la suma de los errores al cuadrado (problema de mínimos cuadrados no lineales):

$$S(\beta) = m \sum i=1 \ (yi - Ae^{-\lambda ti}\cos(\omega ti + \phi))2$$

1. **Base de datos dada**

```r
#-------------------------------------------------------------
# Base de datos
t_data <- c(0.01585558, 0.07313676, 0.20858986, 0.33697873, 0.52537301, 0.6164369,
     0.67744082, 0.77372499, 1.24299318, 1.28798218, 1.38339642, 1.38848687,
     1.42076869, 1.46404696, 1.50057502, 1.62121603, 1.70114607, 1.94117773,
     1.96511879, 2.22784082, 2.27602643, 2.34135203, 2.34449634, 2.41064538,
     2.46308469, 2.49806005, 2.56570644, 2.59826844, 2.61573859, 3.05457166,
     3.28683793, 3.33196052, 3.80886595, 3.84728936, 4.00492237, 4.00529043,
     4.01024845, 4.23964951, 4.38247821, 4.43925851, 4.48290057, 4.56513696,
     4.58015693, 4.61305888, 4.64543839, 4.6630282, 4.73071767, 4.82119210,
     4.93766455, 4.95449219)

y_data <- c(3.30604951, 1.63304262, -2.46669298, -4.67216227,
          -2.65183759, -0.11931208, 1.58435298, 3.20604040, -2.25300509,
          -3.43019723, -4.02265257, -3.46359790, -3.36545235, -2.97468372,
          -2.74451778, 0.12791535, 1.61615275, 2.82364845, 2.71400979,
          -2.18062097, -2.39741066, -2.82599835, -3.27902725, -3.04151353,
          -2.52012235, -2.40448523, -0.90204355, -0.69802758, -0.45049201,
          0.99303212, -2.27801465, -2.64637196, 2.37675322, 2.28167791,
          1.25167305, 1.60197788, 1.19320954, -1.10650603, -2.31930834,
          -1.98134768, -1.55041113, -0.89272659, -0.75329989, -0.09842116,
          -0.05934957, 0.67373312, 0.93814889, 1.54737425, 1.69100012,
          1.18712468)
```

2. .**Derivación para el método de Newton**: Minimizar $S(\beta)$ con el método de Newton requiere su gradiente y su Hessiana. En problemas de mínimos cuadrados, la Hessiana se puede aproximar eficientemente (aproximación de Gauss-Newton) como

$$HS(\beta) \approx 2J^{\top}J,$$

donde J es la matriz Jacobiana del modelo.

- Derive las expresiones para el vector de residuos:
  $ri(\beta) = yi - y(ti, \beta)$
- Derive las expresiones para la matriz Jacobiana:
  $Jij = \partial ri / \partial \beta j$
- Implemente funciones en R o Python que calculen el gradiente $\nabla S(\beta) = -2J^{\top}r$ y la Hessiana aproximada $HS(\beta)$.

3. **Implementación y ejecución:**
    - Adapte su código del método de Newton para usar el gradiente y la Hessiana aproximada que acaba de derivar.
    - Inicie la optimización desde un punto razonable pero no exacto (por ejemplo, βactual = [4.5, 0.3, 6.0, 0.5]).
    - Ejecute el algoritmo para encontrar los parámetros óptimos βestimado.

4. **Análisis y visualización:**
    - Imprima los parámetros estimados y compárelos con los valores verdaderos, para este ejercicio, los valores verdaderos de los parámetros son: A = 5, λ = 0.2, ω = 2π y ϕ =π/4.
    - Cree un gráfico que muestre
        - Los datos ruidosos originales (como un diagrama de dispersión).
        - La curva del modelo con los parámetros verdaderos
        - La curva del modelo ajustado con los parámetros estimados por su algoritmo.
    - Discuta la precisión del ajuste y la eficacia del método para este problema de ingeniería.

Solución:

Función objetivo en lenguaje de R

```r
modelo <- function(A, landa, w, teta, t) {

  A * exp(-landa * t) * cos(w * t + teta)
}


funcion_Objetivo <- function(beta, t_data, y_data) {

  # parametros
  A <- beta[1]
  landa <- beta[2]
  w <- beta[3]
  teta <- beta[4]
  #-------------------------------------------------------------------
  # Nuevo Y o prediccion de Y
  y_pred <- A * exp(-landa * t_data) * cos(w * t_data + teta)
  #---------------------------------------
  #
  sum((y_data - y_pred)^2)
}
```

Sin Jacobiano

Código de Gradiente de la función

```r
gradiente <- function(beta, t_data, y_data) {
  # Parametros
  A <- beta[1]
  landa <- beta[2]
  w <- beta[3]
  teta <- beta[4]
  #---------------------------------------------------------------------
  # Cuantos datos hay en t_data ( un "Len" de python)
  n <- length(t_data)
  #---------------------------------------------------------------------
  # nuevo gradiente
  grad <- numeric(4)

  for(i in 1:n) {
    # rescribiendo t e Y
    t <- t_data[i]
    y <- y_data[i]
    #-----------------------------------------------------------------
    # Predicción
    exp_term <- exp(-landa * t)
    cos_term <- cos(w * t + teta)
    sin_term <- sin(w * t + teta)
    y_pred <- A * exp_term * cos_term
    #-----------------------------------------------------------------
    # Residuo
    r <- y - y_pred
    #-----------------------------------------------------------------
    # Derivadas parciales respecto a cada parámetro

    dy_dA <- exp_term * cos_term                  #deribada de Y segun A
    dy_dlanda <- -A * t * exp_term * cos_term     #deribada de Y segun landa
    dy_dw <- -A * exp_term * sin_term * t         #deribada de Y segun W
    dy_dteta <- -A * exp_term * sin_term          #deribada de Y segun Teta
    #-----------------------------------------------------------------
    # Gradiente (regla de la cadena [NOOO de nuevo jajajjja XD])
    grad[1] <- grad[1] - 2 * r * dy_dA
    grad[2] <- grad[2] - 2 * r * dy_dlanda
    grad[3] <- grad[3] - 2 * r * dy_dw
    grad[4] <- grad[4] - 2 * r * dy_dteta
  }

  return(grad)
}
```

Código de la Matriz Hessiana normal

```r
hessiana <- function(beta, t_data, y_data) {

  # Parametros
  A <- beta[1]
  landa <- beta[2]
  w <- beta[3]
  teta <- beta[4]
  #-----------------------------------------------------------------------
  # Cuantos datos hay en t_data ( un "Len" de python)
  n <- length(t_data)
  #-----------------------------------------------------------------------
  #Matris para la hessiana
  H <- matrix(0, nrow = 4, ncol = 4)

  for(i in 1:n) {
    # rescribiendo t segun la base de datos
    t <- t_data[i]
    #-----------------------------------------------------------------------
    # Predicción
    exp_term <- exp(-landa * t)
    cos_term <- cos(w * t + teta)
    sin_term <- sin(w * t + teta)
    #-----------------------------------------------------------------------
    # Derivadas parciales respecto a cada parámetro
    dy_dA <- exp_term * cos_term                    #deribada de Y segun A
    dy_dlanda <- -A * t * exp_term * cos_term       #deribada de Y segun landa
    dy_dw <- -A * exp_term * sin_term * t           #deribada de Y segun W
    dy_dteta <- -A * exp_term * sin_term            #deribada de Y segun Teta
    #-----------------------------------------------------------------------
    # Derivadas parciales respecto a cada parámetro
    dy_dA <- exp_term * cos_term                    #deribada de Y segun A
    dy_dlanda <- -A * t * exp_term * cos_term       #deribada de Y segun landa
    dy_dw <- -A * exp_term * sin_term * t           #deribada de Y segun W
    dy_dteta <- -A * exp_term * sin_term            #deribada de Y segun Teta
    #-----------------------------------------------------------------------
    # Derivadas de segundo orden (aproximación de Gauss-Newton)
    # H = 2 * J**T * J (ignorando terminos de segundo orden con residuos)
    J <- c(dy_dA, dy_dlanda, dy_dw, dy_dteta)
    H <- H + 2 * outer(J, J)
  }

  return(H)
}
```

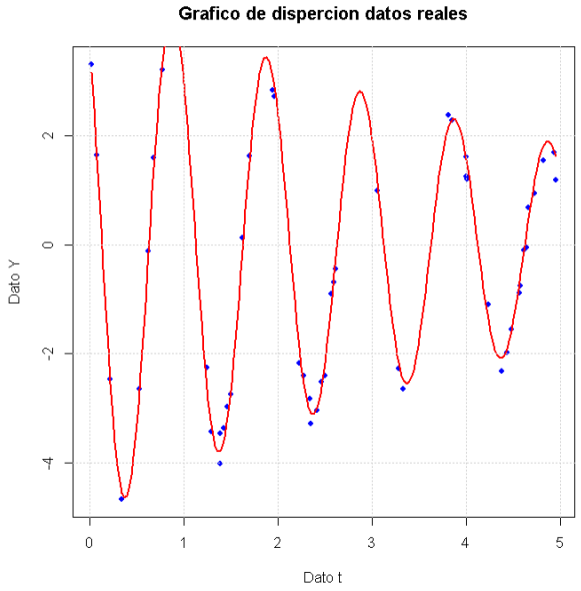
Código de Newton para calcular los nuevos valores de A, Lambda, Omega y Phi

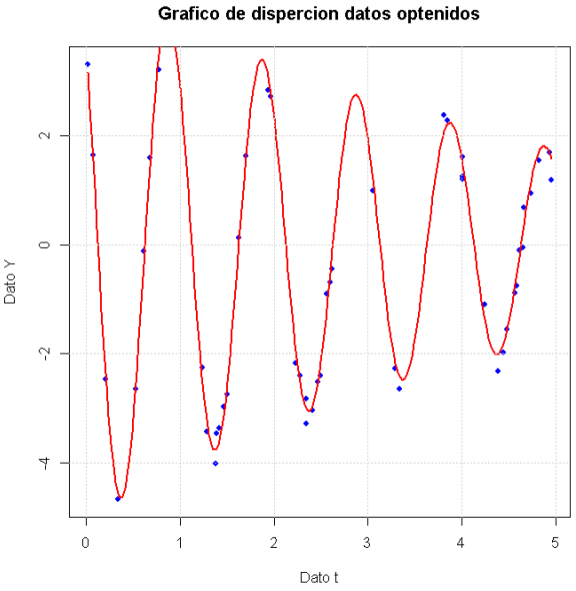
```r
newton <- function(beta_0, t_data, y_data, tol = 1e-6, max_iter = 100) {
  #BETA tiene los valores de A, landa, W y TETA
  beta <- beta_0
  historial <- matrix(NA, nrow = max_iter + 1, ncol = 5)
  colnames(historial) <- c("iter", "A", "landa", "w", "teta")

  historial[1, ] <- c(0, beta)

  for(k in 1:max_iter) {
    grad <- gradiente(beta, t_data, y_data)
    H <- hessiana(beta, t_data, y_data)

    # Resolver H * delta = -grad
    delta <- tryCatch({
      solve(H, -grad)
    }, error = function(e) {
      # Si la Hessiana es singular (signo incorrecto), usar pseudo-inversa
      MASS::ginv(H) %*% (-grad)
    })

    beta_nuevo <- beta + delta
    historial[k + 1, ] <- c(k, beta_nuevo)
```

Datos de los datos obtenidos sin la Matriz Jacobiana

```
=== RESULTADOS Sin JACOBIANO===
> cat("A obtenido     =",resultado$beta[1],"   y A buscado    =",A_real,"\n")
A obtenido     = 5.024849    y A buscado    = 5
> cat("Su precicion es =",precicion_a,"\n")
Su precicion es = 99.50302
> cat("Landa obtenido =",resultado$beta[2],"   y Landa buscad =",landa_rea
l,"\n")
Landa obtenido = 0.2091545    y Landa buscad = 0.2
> cat("Su precicion es =",precicion_landa,"\n")
Su precicion es = 95.42277
> cat("W obtenido     =",resultado$beta[3],"  y W buscado    =",w_real,"\n")
W obtenido     = 6.279533   y W buscado    = 6.283185
> cat("Su precicion es =",precicion_w,"\n")
Su precicion es = 99.94188
> cat("teta obtenido  =",resultado$beta[4],"   y teta buscado =",teta_real,"\n")
teta obtenido  = 0.7909172    y teta buscado = 0.7853982
> cat("Su precicion es =",precicion_teta,"\n")
Su precicion es = 99.2973
> cat("Iteraciones necesarias = ",resultado$iteraciones,"\n")
Iteraciones necesarias =  8
> cat("Valor de la Funcion(β) =", resultado$valor_f, "\n")
Valor de la Funcion(β) = 2.029891
> cat("La precicion del progra es =",precicion_total)
La precicion del progra es = 99.50302
```

Gráficos de los resultados sin Jacobiano





Con Jacobiano

Calculo de Jacobiano

```
#-------------------------------------------------------------------------------
calcular_jacobiana <- function(beta, t_data) {
  # Parametros
  A <- beta[1]
  landa <- beta[2]
  w <- beta[3]
  teta <- beta[4]
  #-----------------------------------------------------------------------------

  # Cuantos datos hay en t_data ( un "Len" de python)
  n <- length(t_data)
  #-----------------------------------------------------------------------------

  # Creacion de la matriz jacobiana
  J <- matrix(0, nrow = n, ncol = 4)

```

```r
  J <- matrix(0, nrow = n, ncol = 4)

  for(i in 1:n) {
    # rescribiendo t e Y
    t <- t_data[i]

    #-------------------------------------------------------------------------
    # Predicción
    exp_term <- exp(-landa * t)
    cos_term <- cos(w * t + teta)
    sin_term <- sin(w * t + teta)

    #-------------------------------------------------------------------------
    # Derivadas parciales respecto a cada parámetro
    J[i, 1] <- -exp_term * cos_term              #deribada de J segun A
    J[i, 2] <- A * t * exp_term * cos_term       #deribada de J segun landa
    J[i, 3] <- A * exp_term * sin_term * t       #deribada de J segun W
    J[i, 4] <- A * exp_term * sin_term           #deribada de J segun Teta

  }
  return(J)
}
#-----------------------------------------------------------------------------
```

Cálculo de Residuo (r)

```r
calcular_r <- function(beta, t_data, y_data) {
  A <- beta[1]
  landa <- beta[2]
  w <- beta[3]
  teta <- beta[4]
  #-------------------------------------------------------------------------

  # Predicciones del modelo
  y_pred <- A * exp(-landa * t_data) * cos(w * t_data + teta)
  #-------------------------------------------------------------------------

  # Vector de residuos
  r <- y_data - y_pred

  return(r)
}
#-----------------------------------------------------------------------------
```

Gradiente de la función con Jacobiano

```r
gradiente <- function(beta, t_data, y_data) {
  # Calcular Jacobiana y residuos
  J <- calcular_jacobiana(beta, t_data)
  r <- calcular_r(beta, t_data, y_data)
  #-------------------------------------------------------------------------

  # Gradiente: ∇S(β) = -2·JᵀT r
  grad <- 2 * t(J) %*% r
  #-------------------------------------------------------------------------

  # Convertir matriz (4×1) a vector
  return(as.vector(grad))
}
#-----------------------------------------------------------------------------
```

Código de la matriz Hessiana con Jacobiano

```r
hessiana <- function(beta, t_data, y_data) {
  # Calcular Jacobiana
  J <- calcular_jacobiana(beta, t_data)
  #----------------------------------------------------------------------

  # Hessiana aproximada: H ≈ 2·JᵀT·J
  H <- 2 * t(J) %*% J

  return(H)
}
#----------------------------------------------------------------------
```

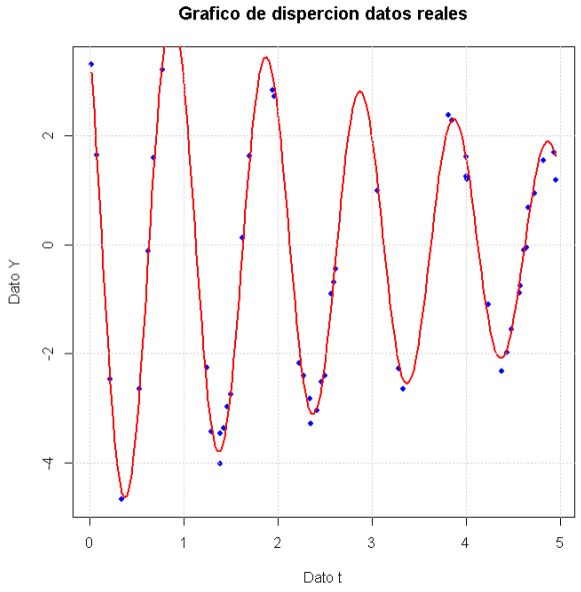
Código de Newton para calcular los nuevos valores de A, Lambda, Omega y Phi

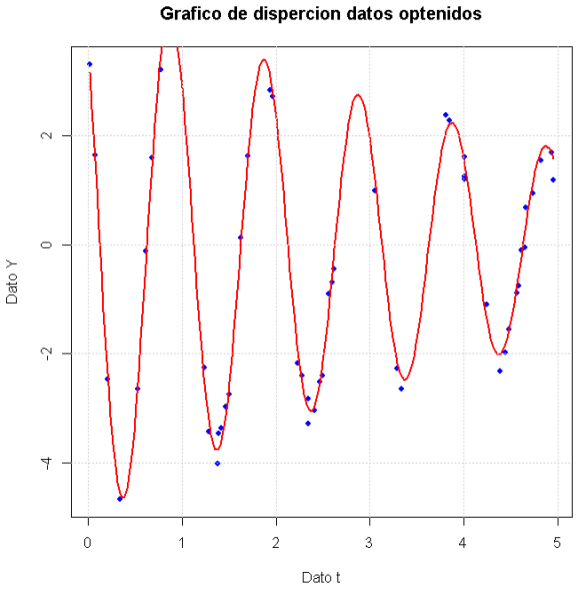
```r
newton <- function(beta_0, t_data, y_data, tol = 1e-6, max_iter = 100) {
  #BETA tiene los valores de A, landa, W y TETA
  beta <- beta_0
  historial <- matrix(NA, nrow = max_iter + 1, ncol = 5)
  colnames(historial) <- c("iter", "A", "landa", "w", "teta")

  historial[1, ] <- c(0, beta)

  for(k in 1:max_iter) {
    grad <- gradiente(beta, t_data, y_data)
    H <- hessiana(beta, t_data, y_data)

    # Resolver H * delta = -grad
    delta <- tryCatch({
      solve(H, -grad)
    }, error = function(e) {
      # Si la Hessiana es singular (signo incorrecto), usar pseudo-inversa
      MASS::ginv(H) %*% (-grad)
    })

    beta_nuevo <- beta + delta
    historial[k + 1, ] <- c(k, beta_nuevo)
```

Datos de los datos obtenidos Con la Matriz Jacobiana

```
=== RESULTADOS Cin JACOBIANO===
> cat("A obtenido     =",resultado$beta[1],"   y A buscado    =",A_real,"\n")
A obtenido     = 5.024849    y A buscado    = 5
> cat("Su precicion es =",precicion_a,"\n")
Su precicion es = 99.50302
> cat("Landa obtenido =",resultado$beta[2],"   y Landa buscad =",landa_rea
l,"\n")
Landa obtenido = 0.2091545    y Landa buscad = 0.2
> cat("Su precicion es =",precicion_landa,"\n")
Su precicion es = 95.42277
> cat("W obtenido     =",resultado$beta[3],"  y W buscado    =",w_real,"\n")
W obtenido     = 6.279533   y W buscado    = 6.283185
> cat("Su precicion es =",precicion_w,"\n")
Su precicion es = 99.94188
> cat("teta obtenido  =",resultado$beta[4],"   y teta buscado =",teta_real,"\n")
teta obtenido  = 0.7909172    y teta buscado = 0.7853982
> cat("Su precicion es =",precicion_teta,"\n")
Su precicion es = 99.2973
> cat("Iteraciones necesarias = ",resultado$iteraciones,"\n")
Iteraciones necesarias =  8
> cat("Valor de la Funcion(β) =", resultado$valor_f, "\n")
Valor de la Funcion(β) = 2.029891
> cat("La precicion del progra es =",precicion_total)
La precicion del progra es = 99.50302
>
```

Gráficos de los datos con Matriz Jacobiano





Por alguna razón los resultados obtenidos a través de la Hessiana sin Jacobiano y la Hessiana con Jacobiano dan los mismos resultados para todos los valores.

	Sin Jacobiano	Con Jacobiano	Teórico
A	5.024849	5.024849	5
Landa	0.2091545	0.2091545	0.2
Omega	6.279533	6.279533	6.283185
Phi	0.7909172	0.7909172	0.7853982
Iteraciones	8	8	—----------
Valor de la función	2.029891	2.029891	—----------

Con esta tabla se puede observar que los datos obtenidos son muy similares a los datos obtenidos.
La precisión obtenida en el programa es de 99.50302%
Como la precisión es alta se puede decir que su eficiencia también es alta

Ejercicio 3

La principal ventaja de los métodos Quasi-Newton es que evitan el cálculo explícito de la Hessiana, lo que los hace ideales para funciones complejas. En este ejercicio, se utilizará el algoritmo BFGS para explorar la superficie de una función con múltiples mínimos y analizar cómo el punto de partida determina el resultado.
Utilice la función de Himmelblau, una prueba clásica para optimización que tiene cuatro mínimos locales idénticos:|

$$f(x,y) = (x\text{^}2 + y - 11)2 + (x + y\text{^}2 - 7)\text{^}2$$

El valor del mínimo de la función es $f(x,y) = 0$. Actividades:

1. 1. Componentes del algoritmo:

    a. Calcule analíticamente el gradiente $\nabla f(x,y)$. En este ejercicio, asumimos que la Hessiana es demasiado compleja o costosa de calcular, justificando el uso de BFGS.

    b. Implemente el algoritmo Quasi-Newton con la actualización BFGS y la búsqueda de paso por retroceso (backtracking).

2. Exploración de los mínimos:

    a. Investigue y encuentre las coordenadas de los cuatro mínimos de la función de Himmelblau.

    b. Ejecute su algoritmo BFGS partiendo de los siguientes tres puntos iniciales:
        i. x0 =(0,0)
        ii. x0 =(−1,3)
        iii. x0 =(4,−2)

3. Análisis y visualización:

    a. Para cada punto de partida, informe a cuál de los cuatro mínimos converge el algoritmo y cuántas iteraciones fueron necesarias.

    b. Genere un único gráfico de contorno de la función de Himmelblau.

    c. En el gráfico, marque la ubicación de los cuatro mínimos teóricos.

    d. Superponga las trayectorias de optimización de sus tres ejecuciones.

    e. Discuta cómo la topografía de la función (sus “valles” y “cuencas de atracción”) guía al algoritmo hacia diferentes soluciones dependiendo de dónde comience.

Solución :

Función de manera código en lenguaje Python

```python
from mpl_toolkits.mplot3d import Axes3D  # Necesario para proyecciones 3D
from matplotlib.widgets import TextBox
import matplotlib.pyplot as plt
import numpy as np
import random
import time
# encontrar la funcion
# f(x,y) = (x^2 + y -11)^2 +(x  + y^2 -7)^2

#funcion inicial respecto a X["x","y"]
def f(x):
    return ((x[0]**2 + x[1] -11)**2 +(x[0] + x[1]**2 -7)**2)
#-----------------------------------------------------------------------------------------------
```

Gradiente

```python
#funcion grandiente de la funcion inicial
def gradiente_f(x):
    #el comando "np.array" sirve para difinir que el valor sera un vector
    #separado por la ","
    dx = ((2*(x[0]**2 + x[1] -11)*(2*x[0])) + 2*(x[0] + x[1]**2 -7))
    dy = (2*(x[0]**2 + x[1] -11)) + (2*(x[0] + x[1]**2 -7)*(2*x[1]))
    return np.array([dx,dy])
##-----------------------------------------------------------------------------------------------
```

Codigo hessiana

```python
#calculkalr H nuevo
def Hk1(x, xk1, gradiente_f, H):
    Sk = xk1 -x                             #calcular Sk= X_nuevo - X_original
    Yk = gradiente_f(xk1) - gradiente_f(x)  #Calcular Yk= Gradeinte de X_nuevo - Gradiente de X_original
    I = np.identity(len(x))                 #I va hacer la matriz identidad
    # Evitar división por cero
    Yk_Sk = np.dot(Yk, Sk)                  #Producto punto entre Yk * Sk
    if abs(Yk_Sk) < 1e-10:                  #si este resultado es muy pequeño conseguimos
        return H                            #el K nuevo
    #-------------------------------------------------------------------------------------------
    # Fórmula BFGS
    term1 = I - np.outer(Sk, Yk) /Yk_Sk
    term2 = I - np.outer(Yk, Sk) /Yk_Sk
    term3 = np.outer(Sk, Sk) /Yk_Sk
    #-------------------------------------------------------------------------------------------
    Hk_1 = term1 @ H @ term2 + term3

    return Hk_1
#-----------------------------------------------------------------------------------------------
```

Codigo Hessiana inversa

```
24
25    #similimar a la matriz hessiana inversa
26    def Pk(Hk, gradiente):
27        # Se usa la multiplicación matricial @
28        pk = -Hk @ gradiente
29        return pk
30    #-------------------------------------------------
```

Codigo para calcular Alpha

```
31
32  #Calcular alpha nuevo
33  def alfa(pk, x, gradiente, funcion_ini):
34      c = 0.0001
35      iteraciones_alpha = 100                 #valor para cambiar alpha si es que esta nunca es suficnetemente menor
36      alpha = 1.0
37      Pk_grad = np.dot(gradiente, pk)         #para el PC es mas facil hacer esta operacion por separado
38      #-----------------------------------------------------------------------------------------
39
40      for _ in range(iteraciones_alpha):
41          n = x + (alpha * pk)                #se necesita un valor tipo vector para usar la funcion
42          auxiliar = f(n)
43          #mientras que:
44              #La funcion inicial segun el vector N   sea mayor a
45                  #funcion inicial original + c * alpha y * el gradiente inversa * Pk
46          if auxiliar <= funcion_ini + (c * alpha * Pk_grad):
47              break                           # Condición cumplida
48          alpha *= 0.5                        # Reducir el paso
49
50      return alpha
51  #-----------------------------------------------------------------------------------------
```

Codigo de BFGS

```
79  #main
80  def BFGS(x_inicial):
81      #el H iniciial es la matriz identidad
82      H = np.identity(2)
83      #-----------------------------------------------------------------------------------------
84      #el valor inicial de X es el punto -10 y +10
85      x = x_inicial
86      max_iteraciones = 1000
87      tolerancia = 1e-6
88      cont=0
89      #-----------------------------------------------------------------------------------------
90      # Almacenar trayectoria
91      trayectoria = [x.copy()]
92
93      for i in range(max_iteraciones):
94          cont=cont+1
95          #-----------------------------------------------------------------------------------
```

```
        #-----------------------------------------------------------------------------------------

        # Calcular función y gradiente
        funcion = f(x)
        gradiente = gradiente_f(x)
        #-----------------------------------------------------------------------------------------

        # Verificar convergencia
        norma_grad = np.linalg.norm(gradiente)
        if norma_grad < tolerancia:
            print(f"Convergencia alcanzada en iteración {i}")
            break
        #-----------------------------------------------------------------------------------------

        #calular cosas nuevas
        pk = Pk(H, gradiente)                       # Calcular dirección de búsqueda
        alpha = alfa(pk, x, gradiente, funcion)     # Búsqueda de línea
        X = x1(x, alpha, pk)                        # Actualizar x
        H = Hk1(x, X, gradiente_f, H)               # Actualizar Hk
        #-----------------------------------------------------------------------------------------

        #el X original sera el valor de X nuevo
        x = X
        trayectoria.append(x.copy())

    return x, trayectoria
```

Codigo para calcular nuevo X

```
#calcular nuevo X
def x1(x,alpha,pk):
    x1= x + (alpha*pk)
    return(x1)
#-------------------------------------
```

Valores obtenidos del primer punto inicial

```
RESULTADO 1
Punto óptimo encontrado: x = [3. 2.]
Valor de la función: f(x) = 0.00000000000
Gradiente final: ∇f(x) = [-3.73510289e-09  6.57180976e-10]
Número de iteraciones: 10
==========
```

Valores obtenidos del segundo punto inicial

```
RESULTADO 2
Punto óptimo encontrado: x = [-2.80511809  3.13131252]
Valor de la función: f(x) = 0.00000000000
Gradiente final: ∇f(x) = [-2.98499896e-10  5.00942683e-11]
Número de iteraciones: 7
==========
```

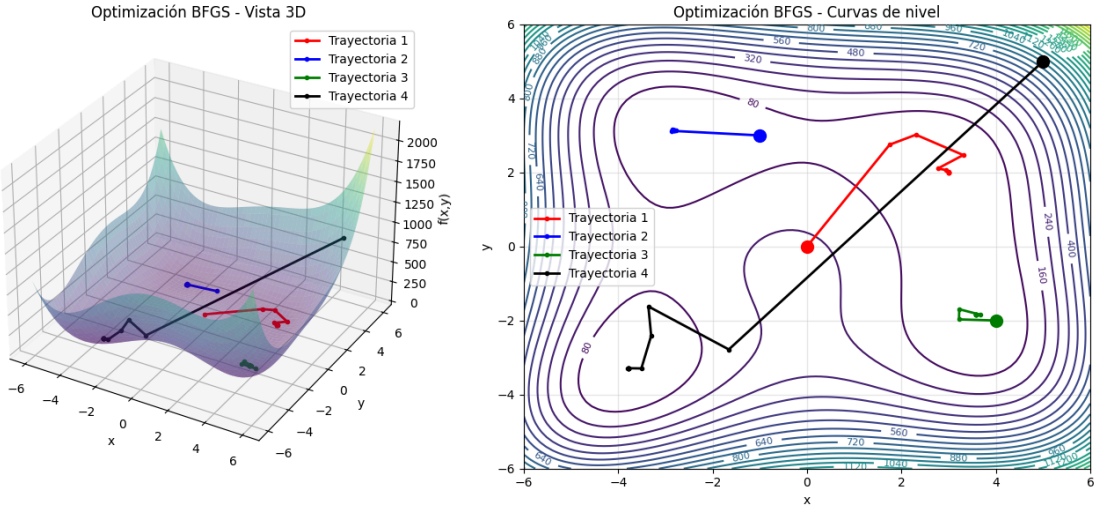
Valores obtenidos del tercer punto inicial

```
RESULTADO 3
Punto óptimo encontrado: x = [ 3.58442834 -1.84812653]
Valor de la función: f(x) = 0.00000000000
Gradiente final: ∇f(x) = [1.70447506e-08 4.05597003e-09]
Número de iteraciones: 9
==========
```

Valores obtenidos del cuarto punto inicial

```
RESULTADO 4
Punto óptimo encontrado: x = [-3.77931025 -3.28318599]
Valor de la función: f(x) = 0.00000000000
Gradiente final: ∇f(x) = [-2.82942214e-07  5.21881036e-07]
Número de iteraciones: 11
```

Gráfico de los puntos iniciales y sus valores



La razón de porque los puntos tienden a sus respectivos mínimos indicados es porque y en especial el punto 4, es porque la función BFGS no busca el mínimo más cercano si el mínimo más estable a través de la predicción de la hessiana y su gradiente, es decir que a través de matriz hessiana estimada se avanza en dirección del gradiente una distancia determinada.

Ejercicio 4

Cuando la matriz Hessiana no es positiva definida, la dirección de búsqueda del método de Newton puede apuntar hacia un máximo o un punto silla, causando que el algoritmo falle. Una solución práctica es la regularización.
Considere la función:

$$f(x,y) = 0.5*x\text{^}2 + 2.5*y\text{^}2 - 2*x*y - x\text{^}3$$

Esta función tiene un mínimo local y un punto silla, lo que la hace interesante para probar la robustez. Actividades:

1. **Análisis de la Hessiana:**
    a. Calcule la matriz Hessiana H$f(x,y)$.
    b. Evalúe la Hessiana en el punto x0 = (1.5,0.5). Calcule sus valores propios (eigenvalues). ¿Es la matriz positiva definida en este punto?
2. **Implementación estándar:**
    a. Ejecuta el método de Newton estándar comenzando desde x0 =(1.5,0.5). Describa lo que sucede. ¿Converge el algoritmo a un mínimo?
3. **Implementación regularizada**
    a. Modifique su algoritmo de Newton para incorporar la técnica de regularización descrita en clases. Específicamente, antes de calcular el paso de actualización, verifique si algún valor propio de la Hessiana es menor o igual a cero. Si es así, modifique la Hessiana sumándole una matriz identidad multiplicada por una constante: Modificada = H + C*I, donde C es un valor pequeño pero suficiente para hacerla positiva definida.
4. **Comparación:**
    a. Ejecuta el algoritmo regularizado desde el mismo punto de partida x0 = (1.5,0.5).
    b. Visualice en un gráfico de contorno las trayectorias del método estándar (si es posible) y del método regularizado.
    c. Explique por qué la regularización permite al algoritmo converger correctamente hacia un mínimo, mientras que la versión estándar falla o se comporta de manera errática.

Solución :

Función en forma de código en lenguaje R

```r
# Definimos la función objetivo
f <- function(x) {
  return(0.5*(x[1]**2)  + 2.5*(x[2]**2) -2*x[1]*x[2]  - x[1]**3 )
}

#-------------------------------------------------------------------------
```

Gradiente

```r
# Definimos el vector gradiente de la función
gradiente_f <- function(x) {
  dx <- x[1] - 2*x[2] - 3*x[1]**2
  dy <- 5*x[2] - 2*x[1]
  return(c(dx, dy))
}

#-------------------------------------------------------------------------
```

Hessiana normal

```r
#-------------------------------------------------------------------------
# Definimos la matriz Hessiana de la función
hessiana_f <- function(x) {
  dxx <- 1 -6*x[1]
  dxy <- -2
  dyx <- -2
  dyy <- 5
  H <- matrix(c(dxx, dxy,
                dyx, dyy ), nrow = 2, byrow = TRUE)
  return(H)
}

```

Hessiana regularizada

```r
# Definimos la matriz Hessiana regularizada de la función
hessiana_R <- function(x, delta = 1) {
  #llamamos a la hesiana normal
  H <- hessiana_f(x)
  #conseguimos los numeros de !!!VALORES PROPIOS!!!
  descomposicion <- eigen(H)

  # Ahora 'delta' está definido dentro del alcance de la función
  #C = Buscame el valor maximo de(DESCOMPOSICION y DEALTA)
  #Evitamos que sea exponencial
  c <- pmax(descomposicion$values, delta)

  I <- diag(nrow(H))

  # Reconstruir la matriz regularizada
  H_reg <- H - c * I
  return(H_reg)
}

```

Cálculo de nuevo X

```r
  # Calculamos el siguiente punto
  paso_newton_R <- solve(hessiana_R(x_k_r)) %*% gradiente_f(x_k_r) #H^-1 * F
  x_k1_r <- x_k_r - paso_newton_R

  # Verificar si hay valores inválidos
  if (any(is.na(x_k1_r)) || any(is.infinite(x_k1_r))) {
    cat("Error: Valores inválidos en iteración", iteraciones_r, "\n")
    cat("x_k1_r =", x_k1_r, "\n")
    break
  }
  #print(paste(x_k1_r))

  #-----------------------------------------------------------------------------
  # Criterios de parada
  if (all(abs(gradiente_f(x_k1_r)) < tol) || all(abs(x_k1_r - x_k_r) < tol)) {
    cat("Hessiana regularizada paró en iteración:", iteraciones_r, "\n")

    # Guardar punto final
    historial_x_R <- c(historial_x_R, x_k1_r[1])
    historial_y_R <- c(historial_y_R, x_k1_r[2])
    historial_f_R <- c(historial_f_R, f(x_k1_r))
    x_k_r <- x_k1_r
    break
  }

  #-----------------------------------------------------------------------------
  x_k_r <- x_k1_r
}
```

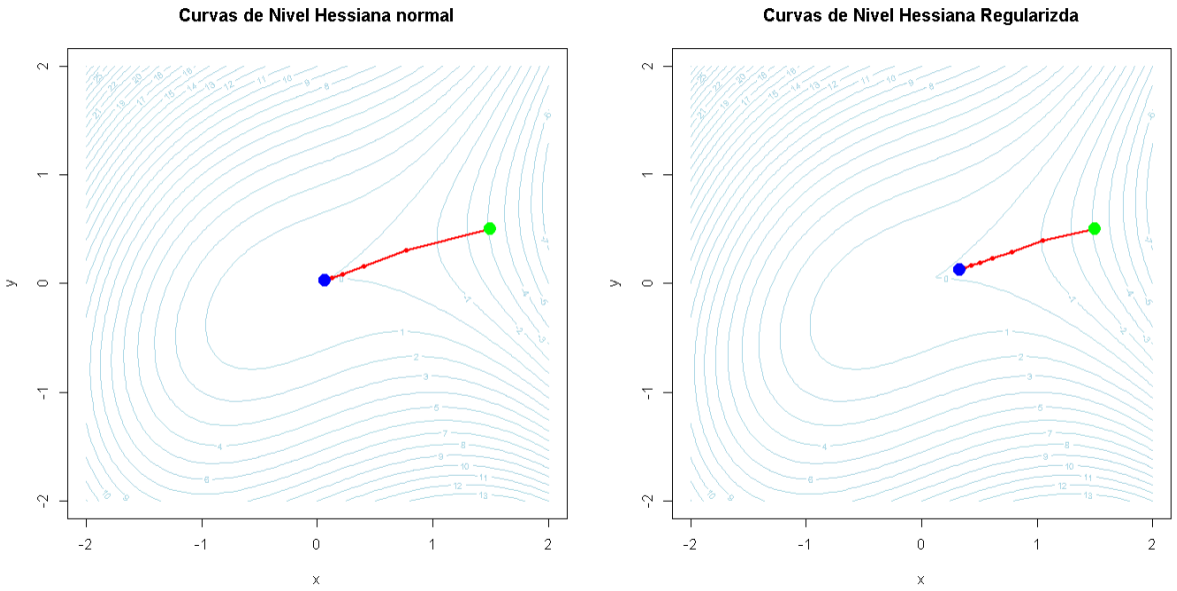
Datos obtenidos

```
Hessiana Normal:
> cat("  Iteraciones:", iteraciones, "\n")
  Iteraciones: 8
> cat("  Mínimo encontrado en x:", x_k_h[1], "\n")
  Mínimo encontrado en x: 0.06686526
> cat("  Precicio del valor X = ",precicion_x_n,"\n")
  Precicio del valor X =  94.92504
> cat("  Mínimo encontrado en y:", x_k_h[2], "\n")
  Mínimo encontrado en y: 0.0267461
> cat("  Precicion del valor Y = ",precicion_y_n,"\n")
  Precicion del valor Y =  95.39379
> cat("  Valor de la función:", f(x_k_h), "\n")
  Valor de la función: 0.0001481442
>
> cat("\nHessiana Regularizada:\n")

Hessiana Regularizada:
> cat("  Iteraciones:", iteraciones_r, "\n")
  Iteraciones: 235
> cat("  Mínimo encontrado en x:", x_k_r[1], "\n")
  Mínimo encontrado en x: 0.06669546
> cat("  Precicion del valor X = ",precicion_x_r, "\n")
  Precicion del valor X =  90.56283
> cat("  Mínimo encontrado en y:", x_k_r[2], "\n")
  Mínimo encontrado en y: 0.0266781
> cat("  Precicion del valor Y = ",precicion_y_r, "\n")
  Precicion del valor Y =  91.38113
> cat("  Valor de la función:", f(x_k_r), "\n")
  Valor de la función: 0.0001481481
>
```

Tabla de diferencia entre Hessiana Normal y Regularizada

	Hessiana normal (N)	Hessiana regularizada ®	Diferencia (R-N)
Iteraciones	8	235	227
Punto X	0,06686526	0,06669546	-0,0001698
Punto Y	0,0267461	0,0266781	-0,000068
F(X,Y)	0,0001481442	0,0001481481	0,0000000039

Gráfico



En ambos casos con hessiana normal y regularizada se obtiene datos muy similares a tal grado que su precisión son las misma y sus puntos mínimos encontrados solo se diferencian por un error del 1e-4 decimales entre ambos métodos con una diferencia significativa el número de iteraciones donde para este programa el método más eficiente vendría siendo el de la hessiana normal