Tarea 12: Diferenciación y minimización de funciones
Métodos Numéricos

Rafael Alejandro García Ramírez

14 de noviembre de 2023

1. Introducción

Conocer la derivada en un punto nos permite sustraer información vítal de un conjunto de datos pues nos permite obtener características que nos permiten describirlo mejor. La importancia de conocer la derivada varía según el sistema que se tenga, por ejemplo, para el conocimiento de la eficacia de un motor de combustión conocer las primeras, segundas y derivadas de orden mayor es fundamental al momento de la evaluación de este. Sobra decir que existen dos escenarios diferentes: cuando se conoce la función pero la derivada es difícil de evaluar analíticamente y cuando solamente se posee un conjunto de datos (como suele ser el caso de la mayoría de las aplicaciones).

Con lo anterior dicho, debemos extendernos en el hecho de la dificultad que existe al momento de encontrar esta información en algunos sistemas, pues si bien existe una aproximación sencilla que se trata de realizar modelos completamente lineales, no se puede ignorar el hecho de la existencia de sistemas no lineales cuyo comportamiento es totalmente diferente y que cuya modelación con diferentes bases resulta más exacto a los datos que se recolectan.

A continuación se presenta la implementación de varios métodos para la diferenciación de funciones y datos, junto con la resolución de sistemas no lineales.

2. Pseudocódigos

A continuación se presentan los pseudocódigos utilizados en este trabajo, en el caso de los métodos se adjuntó el pseudocódigo para dos variables, aunque la implementación para tres variables es completamente análogo:

2.1. Derivadas para funciones

2.1.1. Derivada hacia adelante

Algorithm 1 Derivada hacia Adelante

function DerivadaHaciaAdelante(f, x, h)
    return $(f(x+h) - f(x))/h$
end function

2.1.2. Derivada hacia atrás

Algorithm 2 Derivada hacia Atrás

function DerivadaHaciaAtras(f, x, h)
    return $(f(x) - f(x-h))/h$
end function

2.1.3. Derivada centrada

Algorithm 3 Derivada Centrada

function DerivadaCentrada(f, x, h)
    return $(f(x+h) - f(x-h))/(2h)$
end function

2.1.4. Derivada tres puntos endpoint

Algorithm 4 Derivada de Tres Puntos (Endpoint)

1: **function** DerivadaTresPuntosEndpoint(f, x, h)
2: **return** $(1/(2h)) \cdot (-3f(x) + 4f(x+h) - f(x+2h))$
3: **end function**

2.1.5. Derivada tres puntos midpoint

Algorithm 5 Derivada de Tres Puntos (Midpoint)

1: **function** DerivadaTresPuntosMidpoint(f, x, h)
2: **return** $(1/(2h)) \cdot (f(x+h) - f(x-h))$
3: **end function**

2.1.6. Derivada cinco puntos endpoint

Algorithm 6 Derivada de Cinco Puntos (Endpoint)

1: **function** DerivadaCincoPuntosEndpoint(f, x, h)
2: **return** $(1/(12h)) \cdot (-25f(x) + 48f(x+h) - 36f(x+2h) + 16f(x+3h) - 3f(x+4h))$
3: **end function**

2.1.7. Derivada cinco puntos midpoint

Algorithm 7 Derivada de Cinco Puntos (Midpoint)

1: **function** DerivadaCincoPuntosMidpoint(f, x, h)
2: **return** $(1/(12h)) \cdot (f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h))$
3: **end function**

2.1.8. Segunda derivada midpoint

Algorithm 8 Segunda Derivada (Midpoint)

1: **function** SegundaDerivadaMidpoint(f, x, h)
2: **return** $(1/(h^2)) \cdot (f(x-h) - 2f(x) + f(x+h))$
3: **end function**

2.2. Para un conjunto de datos

2.2.1. Derivada hacia adelante con datos

Algorithm 9 Derivada hacia Adelante para Datos

1: **function** DerivadaHaciaAdelanteDatos($f1, f2, h$)
2: **return** $(f2 - f1)/h$
3: **end function**

2.2.2. Derivada hacia atrás con datos

Algorithm 10 Derivada hacia Atrás para Datos

1: **function** DerivadaHaciaAtrasDatos($f1, f2, h$)
2: **return** $(f2 - f1)/h$
3: **end function**

2.2.3. Derivada centrada para datos

Algorithm 11 Derivada Centrada para Datos

function DerivadaCentradaDatos($f1, f2, h$)
    return $(f2 - f1)/(2h)$
end function

2.2.4. Derivada tres puntos endpoint para datos

Algorithm 12 Derivada de Tres Puntos (Endpoint) para Datos

function DerivadaTresPuntosEndpointDatos($f1, f2, f3, h$)
    return $(1/(2h)) \cdot (-3f1 + 4f2 - f3)$
end function

2.2.5. Derivada tres puntos midpoint para datos

Algorithm 13 Derivada de Tres Puntos (Midpoint) para Datos

function DerivadaTresPuntosMidpointDatos($f1, f2, h$)
    return $(1/(2h)) \cdot (f1 - f2)$
end function

2.2.6. Derivada cinco puntos endpoint para datos

Algorithm 14 Derivada de Cinco Puntos (Endpoint) para Datos

function DerivadaCincoPuntosEndpointDatos($f1, f2, f3, f4, f5, h$)
    return $(1/(12h)) \cdot (-25f1 + 48f2 - 36f3 + 16f4 - 3f5)$
end function

2.2.7. Derivada cinco puntos midpoint para datos

Algorithm 15 Derivada de Cinco Puntos (Midpoint) para Datos

function DerivadaCincoPuntosMidpointDatos($f1, f2, f3, f4, h$)
    return $(1/(12h)) \cdot (f1 - 8f2 + 8f3 - f4)$
end function

2.2.8. Segunda derivada para datos

Algorithm 16 Segunda Derivada (Midpoint) para Datos

function SegundaDerivadaMidpointDatos($f1, f2, f3, h$)
    return $(1/(h^2)) \cdot (f1 - 2f2 + f3)$
end function

2.2.9. Jacobiano 3 variables

Algorithm 17 Jacobiano de 3 variables

procedure Jacobiano3($J, x, y, z, h, f1, f2, f3$)
    $J[0][0] \leftarrow (1/(12h)) \cdot (-25f1(x,y,z) + 48f1(x+h,y,z) - 36f1(x+2h,y,z) + 16f1(x+3h,y,z) - 3f1(x+4h,y,z))$
    $J[0][1] \leftarrow (1/(12h)) \cdot (-25f1(x,y,z) + 48f1(x,y+h,z) - 36f1(x,y+2h,z) + 16f1(x,y+3h,z) - 3f1(x,y+4h,z))$
    $J[0][2] \leftarrow (1/(12h)) \cdot (-25f1(x,y,z) + 48f1(x,y,z+h) - 36f1(x,y,z+2h) + 16f1(x,y,z+3h) - 3f1(x,y,z+4h))$
    $J[1][1] \leftarrow (1/(12h)) \cdot (-25f2(x,y,z) + 48f2(x,y+h,z) - 36f2(x,y+2h,z) + 16f2(x,y+3h,z) - 3f2(x,y+4h,z))$
    $J[2][2] \leftarrow (1/(12h)) \cdot (-25f3(x,y,z) + 48f3(x,y,z+h) - 36f3(x,y,z+2h) + 16f3(x,y,z+3h) - 3f3(x,y,z+4h))$
    $J[1][2] \leftarrow (1/(12h)) \cdot (-25f2(x,y,z) + 48f2(x,y,z+h) - 36f2(x,y,z+2h) + 16f2(x,y,z+3h) - 3f2(x,y,z+4h))$
    $J[2][1] \leftarrow (1/(12h)) \cdot (-25f3(x,y,z) + 48f3(x,y+h,z) - 36f3(x,y+2h,z) + 16f3(x,y+3h,z) - 3f3(x,y+4h,z))$
    $J[1][0] \leftarrow (1/(12h)) \cdot (-25f2(x,y,z) + 48f2(x+h,y,z) - 36f2(x+2h,y,z) + 16f2(x+3h,y,z) - 3f2(x+4h,y,z))$
    $J[2][0] \leftarrow (1/(12h)) \cdot (-25f3(x,y,z) + 48f3(x+h,y,z) - 36f3(x+2h,y,z) + 16f3(x+3h,y,z) - 3f3(x+4h,y,z))$
end procedure

2.2.10. Jacobiano 2 variables

Algorithm 18 Jacobiano de 2 variables

procedure JACOBIANO2($J, x, y, h, f1, f2$)
  $J[0][0] \leftarrow (1/(12h)) \cdot (-25f1(x,y) + 48f1(x+h,y) - 36f1(x+2h,y) + 16f1(x+3h,y) - 3f1(x+4h,y))$
  $J[0][1] \leftarrow (1/(12h)) \cdot (-25f1(x,y) + 48f1(x,y+h) - 36f1(x,y+2h) + 16f1(x,y+3h) - 3f1(x,y+4h))$
  $J[1][1] \leftarrow (1/(12h)) \cdot (-25f2(x,y) + 48f2(x,y+h) - 36f2(x,y+2h) + 16f2(x,y+3h) - 3f2(x,y+4h))$
  $J[1][0] \leftarrow (1/(12h)) \cdot (-25f2(x,y) + 48f2(x+h,y) - 36f2(x+2h,y) + 16f2(x+3h,y) - 3f2(x+4h,y))$
end procedure

2.2.11. Hessiana 3 variables

Algorithm 19 Hessiana de 3 variables

procedure HESSIANA3($H, x, y, z, h, f1, f2, f3$)
  $H[0][0] \leftarrow (1/(h^2)) \cdot (f1(x+h,y,z) - 2f1(x,y,z) + f1(x-h,y,z))$
  $H[0][1] \leftarrow (1/(h^2)) \cdot (f1(x+h,y+h,z) - 2f1(x,y+h,z) + f1(x-h,y+h,z) - 2f1(x+h,y,z) + 4f1(x,y,z) - 2f1(x-h,y,z) + f1(x+h,y-h,z) - 2f1(x,y-h,z) + f1(x-h,y-h,z))$
  $H[0][2] \leftarrow (1/(h^2)) \cdot (f1(x+h,y,z+h) - 2f1(x,y,z+h) + f1(x-h,y,z+h) - 2f1(x+h,y,z) + 4f1(x,y,z) - 2f1(x-h,y,z) + f1(x+h,y,z-h) - 2f1(x,y,z-h) + f1(x-h,y,z-h))$
  $H[1][0] \leftarrow H[0][1]$
  $H[1][1] \leftarrow (1/(h^2)) \cdot (f2(x+h,y,z) - 2f2(x,y,z) + f2(x-h,y,z))$
  $H[1][2] \leftarrow (1/(h^2)) \cdot (f2(x,y+h,z) - 2f2(x,y,z) + f2(x,y-h,z))$
  $H[2][0] \leftarrow H[0][2]$
  $H[2][1] \leftarrow H[1][2]$
  $H[2][2] \leftarrow (1/(h^2)) \cdot (f3(x+h,y,z) - 2f3(x,y,z) + f3(x-h,y,z))$
end procedure

2.2.12. Hessiana 2 variables

Algorithm 20 Hessiana de 2 variables

procedure HESSIANA2($H, x, y, h, f1, f2$)
  $H[0][0] \leftarrow (1/(h^2)) \cdot (f1(x+h,y) - 2f1(x,y) + f1(x-h,y))$
  $H[0][1] \leftarrow (1/(h^2)) \cdot (f1(x+h,y+h) - 2f1(x,y+h) + f1(x-h,y+h) - 2f1(x+h,y) + 4f1(x,y) - 2f1(x-h,y) + f1(x+h,y-h) - 2f1(x,y-h) + f1(x-h,y-h))$
  $H[1][0] \leftarrow H[0][1]$
  $H[1][1] \leftarrow (1/(h^2)) \cdot (f2(x+h,y) - 2f2(x,y) + f2(x-h,y))$
end procedure

2.3. Método punto fijo no lineal

Algorithm 21 Punto Fijo No Lineal

```
1: procedure PuntoFijoNoLineal(N, x0, tolerancia, iteraciones, f1, f2)
2:     i_convergencia ← 0
3:     x1_anterior, x2_anterior ← 0
4:     while iteraciones > 0 do
5:         for j ← 0 to N − 1 do
6:             x1_anterior ← x0[0]
7:             x2_anterior ← x0[1]
8:             x0[0] ← f1(x0[0], x0[1])
9:             x0[1] ← f2(x0[0], x0[1])
10:        end for
11:        i_convergencia ← i_convergencia + 1
12:        if |x0[0] − x1_anterior| < tolerancia and |x0[1] − x2_anterior| < tolerancia then
13:            Imprimir "Punto Fijo no lineal: convergencia alcanzada en i_convergencia iteraciones"
14:            Devolver
15:        end if
16:        iteraciones ← iteraciones − 1
17:    end while
18:    Imprimir "Punto Fijo no lineal: No converge en i_convergencia iteraciones"
19: end procedure
```

2.4. Método de Newton para ecuaciones no lineales

Algorithm 22 Newton No Lineal

```
1: procedure NewtonNoLineal(N, J, F, x0, tolerancia, h, iteraciones, f1, f2)
2:     i_convergencia ← 0
3:     x_anterior ← generar_vector(N)
4:     dummy ← 0
5:     while iteraciones > 0 do
6:         F[0] ← f1(x0[0], x0[1])
7:         F[1] ← f2(x0[0], x0[1])
8:         LLamar a jacobiano2(J, x0[0], x0[1], h, f1, f2)
9:         dummy ← −J[1][0]/J[0][0]
10:        J[1][1] ← J[1][1] − dummy · J[0][1]
11:        F[1] ← F[1] − dummy · F[0]
12:        x_anterior[1] ← F[1]/J[1][1]
13:        x_anterior[0] ← (F[0] − J[0][1] · x_anterior[1])/J[0][0]
14:        x0[0] ← x0[0] + x_anterior[0]
15:        x0[1] ← x0[1] + x_anterior[1]
16:        if |x_anterior[0]| < tolerancia and |x_anterior[1]| < tolerancia then
17:            Imprimir "Newton no lineal: convergencia alcanzada en i_convergencia iteraciones"
18:            Liberar memoria x_anterior
19:            Devolver
20:        end if
21:        i_convergencia ← i_convergencia + 1
22:        iteraciones ← iteraciones − 1
23:    end while
24:    Imprimir "Newton no lineal: no converge en i_convergencia iteraciones"
25:    Liberar memoria x_anterior
26: end procedure
```

2.5. Método de Broyden

Algorithm 23 Broyden

```
1: procedure BROYDEN(x, tolerancia, iteraciones, f1, f2)
2:     i ← 0
3:     F[2], s[2], u[2], v[2], w[2], yk[2], z[2] ← 0
4:     A[2][2], dummy ← 0
5:     A[0][0] ← 1.0, A[0][1] ← 1.0
6:     A[1][0] ← 2.0, A[1][1] ← 2.0
7:     while iteraciones > 0 do
8:         w[0] ← v[0]
9:         w[1] ← v[1]
10:        F[0] ← f1(x[0], x[1])
11:        F[1] ← f2(x[0], x[1])
12:        v[0] ← F[0]
13:        v[1] ← F[1]
14:        yk[0] ← v[0] − w[0]
15:        yk[1] ← v[1] − w[1]
16:        z[0] ← −A[0][0] · yk[0] − A[0][1] · yk[1]
17:        z[1] ← −A[1][0] · yk[0] − A[1][1] · yk[1]
18:        u[0] ← s[0] · A[0][0] + s[1] · A[1][0]
19:        u[1] ← s[0] · A[0][1] + s[1] · A[1][1]
20:        dummy ← u[0] · z[0] + u[1] · z[1]
21:        i ← i + 1
22:        if |dummy| < tolerancia then
23:            Imprimir Error Broyden: matriz no invertible. El método diverge.
24:            Devolver
25:        end if
26:        A[0][0] ← A[0][0] + (s[0] + z[0]) · u[0]/dummy
27:        A[0][1] ← A[0][1] + (s[0] + z[0]) · u[1]/dummy
28:        A[1][0] ← A[1][0] + (s[1] + z[1]) · u[0]/dummy
29:        A[1][1] ← A[1][1] + (s[1] + z[1]) · u[1]/dummy
30:        s[0] ← −A[0][0] · F[0] − A[0][1] · F[1]
31:        s[1] ← −A[1][0] · F[0] − A[1][1] · F[1]
32:        x[0] ← x[0] + s[0]
33:        x[1] ← x[1] + s[1]
34:        if |s[0]| < tolerancia and |s[1]| < tolerancia then
35:            Imprimir "Método Broyden: convergencia alcanzada en i iteraciones"
36:            Devolver
37:        end if
38:        iteraciones ← iteraciones − 1
39:    end while
40:    Imprimir Error Broyden: El método no converge después de i iteraciones.
41: end procedure
```

2.6. Método de gradiente conjugado de Fletcher-Reeves

Algorithm 24 Gradiente Conjugado No Lineal

```
1: procedure GradienteConjugadoNoLineal(x, tolerancia, iteraciones, f_1, f_2)
2:     i ← 0
3:     F[0] ← f_1(x[0], x[1])
4:     F[1] ← f_2(x[0], x[1])
5:     g[0] ← F[0]
6:     g[1] ← F[1]
7:     g_anterior[0] ← g[0]
8:     g_anterior[1] ← g[1]
9:     d[0] ← −g[0]
10:    d[1] ← −g[1]
11:    while iteraciones > 0 do
12:        alpha ← 0.1
13:        x[0] ← x[0] + alpha × d[0]
14:        x[1] ← x[1] + alpha × d[1]
15:        F[0] ← f_1(x[0], x[1])
16:        F[1] ← f_2(x[0], x[1])
17:        g[0] ← F[0]
18:        g[1] ← F[1]
19:        if i > 0 then
20:            beta ← (g[0]^2 + g[1]^2)/(g_anterior[0]^2 + g_anterior[1]^2)
21:            d[0] ← −g[0] + beta × d_anterior[0]
22:            d[1] ← −g[1] + beta × d_anterior[1]
23:        end if
24:        if fabs(g[0]) < tolerancia and fabs(g[1]) < tolerancia then
25:            Imprimir "Gradiente Conjugado NoLineal: Convergencia alcanzada en i iteraciones"
26:            Devolver
27:        end if
28:        g_anterior[0] ← g[0]
29:        g_anterior[1] ← g[1]
30:        d_anterior[0] ← d[0]
31:        d_anterior[1] ← d[1]
32:        i ← i + 1
33:        iteraciones ← iteraciones − 1
34:    end while
35:    Imprimir Error Gradiente Conjugado NoLineal: No convergió después de i iteraciones
36: end procedure
```

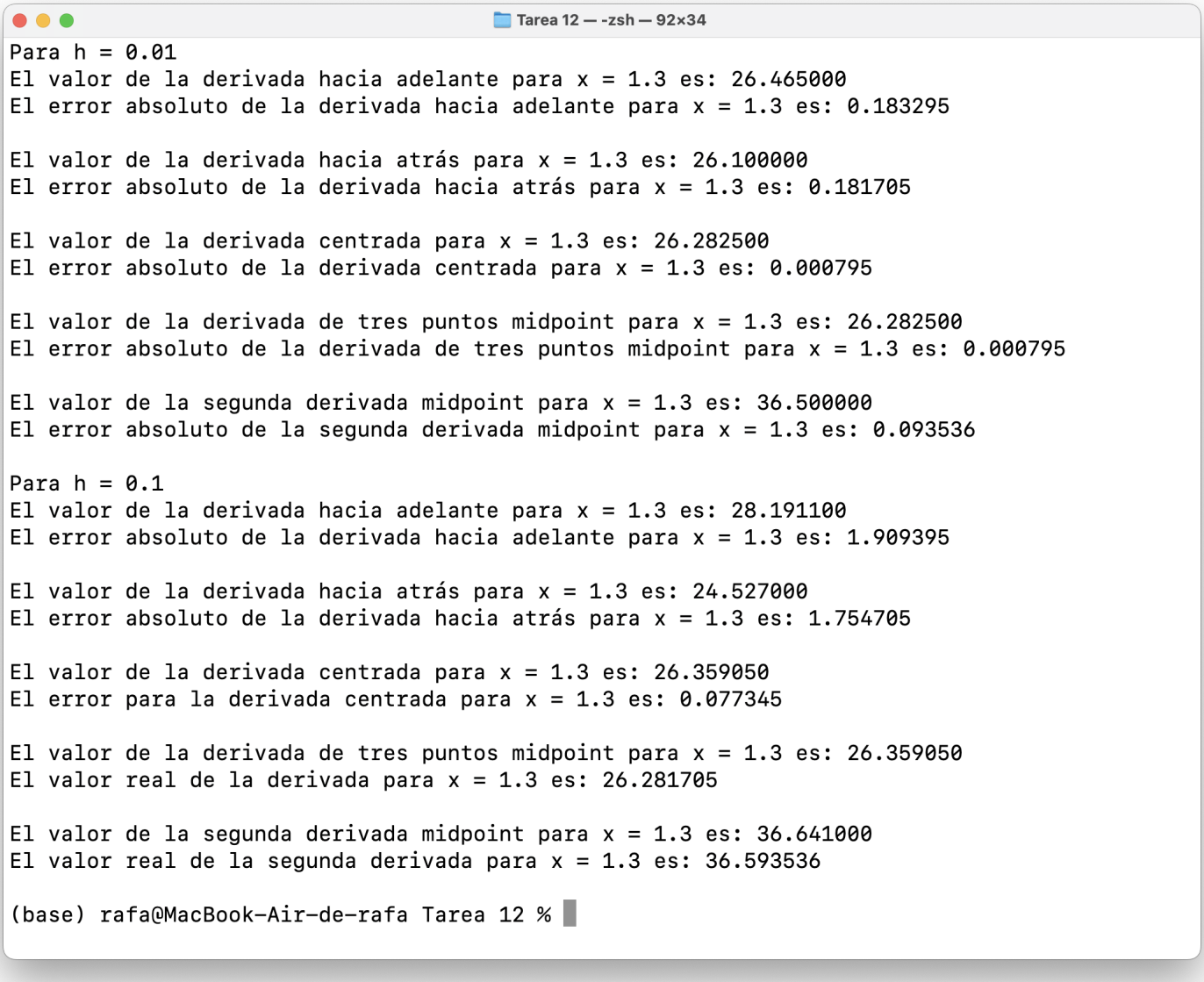
3. Resultados

1. Creamos los siguientes algoritmos en `C` para las diferentes derivadas

```c
#include <stdio.h>

double derivada_hacia_adelante(double (*f)(double), double x, double h){
    return (f(x+h)-f(x))/h;
}
double derivada_hacia_atras(double (*f)(double), double x, double h){
    return (f(x)-f(x-h))/h;
}
double derivada_centrada(double (*f)(double), double x, double h){
    return (f(x+h)-f(x-h))/(2*h);
}
double derivada_tres_puntos_endpoint(double (*f)(double), double x, double h){
    return (1/(2*h))*(-3*f(x)+4*f(x+h)-f(x+2*h));
}
double derivada_tres_puntos_midpoint(double (*f)(double), double x, double h){
    return (1/(2*h))*(f(x+h)-f(x-h));
}
double derivada_cinco_puntos_midpoint(double (*f)(double), double x, double h){
    return (1/(12*h))*(f(x-2*h)-8*f(x-h)+8*f(x+h)-f(x+2*h));
}
double derivada_cinco_puntos_endpoint(double (*f)(double), double x, double h){
    return (1/(12*h))*(-25*f(x) + 48*f(x+h) - 36*f(x+2*h) + 16*f(x+3*h) - 3*f(x+4*h));
}
double segunda_derivada(double (*f)(double), double x, double h){
    return (1/(h*h))*(f(x-h)-2*f(x)+f(x+h));
}
double derivada_hacia_adelante_datos(double f1, double f2, double h){
    // f1 es el dato en f(x)
    // f2 es el dato en f(x+h)
    return (f2-f1)/h;
}
double derivada_hacia_atras_datos(double f1, double f2, double h){
    // f1 es el dato en f(x-h)
    // f2 es el dato en f(x)
    return (f2-f1)/h;
}
double derivada_centrada_datos(double f1, double f2, double h){
    // f1 es el dato en f(x-h)
    // f2 es el dato en f(x+h)
    return (f2-f1)/(2*h);
}
double derivada_tres_puntos_endpoint_datos(double f1, double f2, double f3, double h){
    // f1 es el dato en f(x)
    // f2 es el dato en f(x+h)
    // f3 es el dato en f(x+2h)
    return (1/(2*h))*(-3*f1+4*f2-f3);
}
double derivada_tres_puntos_midpoint_datos(double f1, double f2, double h){
    // f1 es el dato en f(x+h)
    // f2 es el dato en f(x-h)
    return (1/(2*h))*(f1-f2);
}
double derivada_cinco_puntos_midpoint_datos( double f1, double f2, double f3, double f4, double h)
    {
    // f1 es el dato en f(x-2h)
    // f2 es el dato en f(x-h)
    // f3 es el dato en f(x+h)
    // f4 es el dato en f(x+2h)
    return (1/12*h)*(f1-8*f2+8*f3-f4);
}
double derivada_cinco_puntos_endpoint_datos(double f1, double f2, double f3, double f4, double f5,
     double h){
    // f1 es el dato en f(x)
    // f2 es el dato en f(x+h)
    // f3 es el dato en f(x+2h)
    // f4 es el dato en f(x+3h)
    // f5 es el dato en f(x+4h)
    return (1/(12*h))*(-25*f1+48*f2-36*f3+16*f4-3*f5);
}
double segunda_derivada_midpoint_datos(double f1, double f2, double f3, double h){
    // f1 es el dato en f(x-h)
    // f2 es el dato en f(x)
    // f3 es el dato en f(x+h)
    return (1/(h*h))*(f1-2*f2+f3);
}
```

Con los datos que se nos proporcionan, los métodos que se pueden se pueden aplicar y sus resultados son los siguientes



2. Para el jacobiano tomamos la función con dos variables

$$f(x,y) = (x^4 + 3y^2x, 5y^2 - 2xy + 1)$$

Y la función para tres variables

$$g(x,y,z) = (x^4 + 3y^2x, 5y^2 - 2xy + 1, x + y + z)$$

Cuyos jacobianos son para $x = 1.0$ y $y = 2.0$ y $z = 3.0$ los siguientes

$$J_f = \begin{pmatrix} 16 & 12 \\ -4 & 18 \end{pmatrix}$$

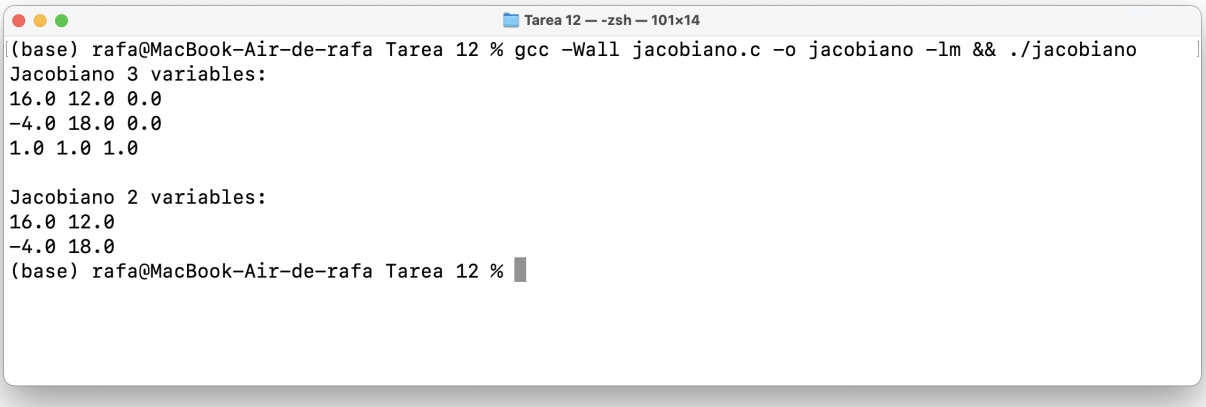
$$J_g = \begin{pmatrix} 16 & 12 & 0 \\ -4 & 18 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Para determinar el jacobiano en cada caso para las diferentes dimensiones creamos las siguientes funciones

```c
#include <stdio.h>

void jacobiano3(double **J, double x, double y, double z, double h, double (*f1)(double,double,
    double), double (*f2)(double,double,double), double (*f3)(double,double,double)){
    J[0][0] = (1/(12*h))*(-25*f1(x,y,z) + 48*f1(x+h,y,z) - 36*f1(x+2*h,y,z) + 16*f1(x+3*h,y,z) -
    3*f1(x+4*h,y,z));
    J[0][1] = (1/(12*h))*(-25*f1(x,y,z) + 48*f1(x,y+h,z) - 36*f1(x,y+2*h,z) + 16*f1(x,y+3*h,z) -
    3*f1(x,y+4*h,z));
    J[0][2] = (1/(12*h))*(-25*f1(x,y,z) + 48*f1(x,y,z+h) - 36*f1(x,y,z+2*h) + 16*f1(x,y,z+3*h) -
    3*f1(x,y,z+4*h));
    J[1][1] = (1/(12*h))*(-25*f2(x,y,z) + 48*f2(x,y+h,z) - 36*f2(x,y+2*h,z) + 16*f2(x,y+3*h,z) -
    3*f2(x,y+4*h,z));
    J[2][2] = (1/(12*h))*(-25*f3(x,y,z) + 48*f3(x,y,z+h) - 36*f3(x,y,z+2*h) + 16*f3(x,y,z+3*h) -
    3*f3(x,y,z+4*h));
    J[1][2] = (1/(12*h))*(-25*f2(x,y,z) + 48*f2(x,y,z+h) - 36*f2(x,y,z+2*h) + 16*f2(x,y,z+3*h) -
    3*f2(x,y,z+4*h));
    J[2][1] = (1/(12*h))*(-25*f3(x,y,z) + 48*f3(x,y+h,z) - 36*f3(x,y+2*h,z) + 16*f3(x,y+3*h,z) -
    3*f3(x,y+4*h,z));
    J[1][0] = (1/(12*h))*(-25*f2(x,y,z) + 48*f2(x+h,y,z) - 36*f2(x+2*h,y,z) + 16*f2(x+3*h,y,z) -
    3*f2(x+4*h,y,z));
    J[2][0] = (1/(12*h))*(-25*f3(x,y,z) + 48*f3(x+h,y,z) - 36*f3(x+2*h,y,z) + 16*f3(x+3*h,y,z) -
    3*f3(x+4*h,y,z));
}
void jacobiano2(double **J, double x, double y, double h, double (*f1)(double,double), double (*f2
    )(double,double)){
    J[0][0] = (1/(12*h))*(-25*f1(x,y) + 48*f1(x+h,y) - 36*f1(x+2*h,y) + 16*f1(x+3*h,y) - 3*f1(x+4*
    h,y));
    J[0][1] = (1/(12*h))*(-25*f1(x,y) + 48*f1(x,y+h) - 36*f1(x,y+2*h) + 16*f1(x,y+3*h) - 3*f1(x,y
    +4*h));
    J[1][1] = (1/(12*h))*(-25*f2(x,y) + 48*f2(x,y+h) - 36*f2(x,y+2*h) + 16*f2(x,y+3*h) - 3*f2(x,y
    +4*h));
    J[1][0] = (1/(12*h))*(-25*f2(x,y) + 48*f2(x+h,y) - 36*f2(x+2*h,y) + 16*f2(x+3*h,y) - 3*f2(x+4*
    h,y));
}
```

Con $h = 0.001$ obtenemos los siguientes resultados



Por lo que podemos confirmar los resultados para el jacobiano. Para la hessiana se creó el siguiente código

```c
#include <stdio.h>

void hessiana3(double **H, double x, double y, double z, double h,  double (*f)(double, double,
    double)) {
    H[0][0] = (f(x + h, y, z) - 2 * f(x, y, z) + f(x - h, y, z)) / (h * h);
    H[1][1] = (f(x, y + h, z) - 2 * f(x, y, z) + f(x, y - h, z)) / (h * h);
    H[2][2] = (f(x, y, z + h) - 2 * f(x, y, z) + f(x, y, z - h)) / (h * h);

    H[0][1] = H[1][0] = (f(x + h, y + h, z) - f(x + h, y - h, z) - f(x - h, y + h, z) + f(x - h, y
     - h, z)) / (4 * h * h);
    H[0][2] = H[2][0] = (f(x + h, y, z + h) - f(x + h, y, z - h) - f(x - h, y, z + h) + f(x - h, y
    , z - h)) / (4 * h * h);
    H[1][2] = H[2][1] = (f(x, y + h, z + h) - f(x, y + h, z - h) - f(x, y - h, z + h) + f(x, y - h
    , z - h)) / (4 * h * h);
}

void hessiana2(double **H, double x, double y, double h,  double (*f)(double, double)) {
    H[0][0] = (f(x + h, y) - 2 * f(x, y) + f(x - h, y)) / (h * h);
    H[1][1] = (f(x, y + h) - 2 * f(x, y) + f(x, y - h)) / (h * h);
```

```
    H[0][1] = H[1][0] = (f(x + h, y + h) - f(x + h, y - h) - f(x - h, y + h) + f(x - h, y - h)) /
    (4 * h * h);
}
```

Con las siguientes funciones

$$f(x,y) = x^2y + y^2x$$

$$g(x,y,z) = e^{-x}\sin xy$$

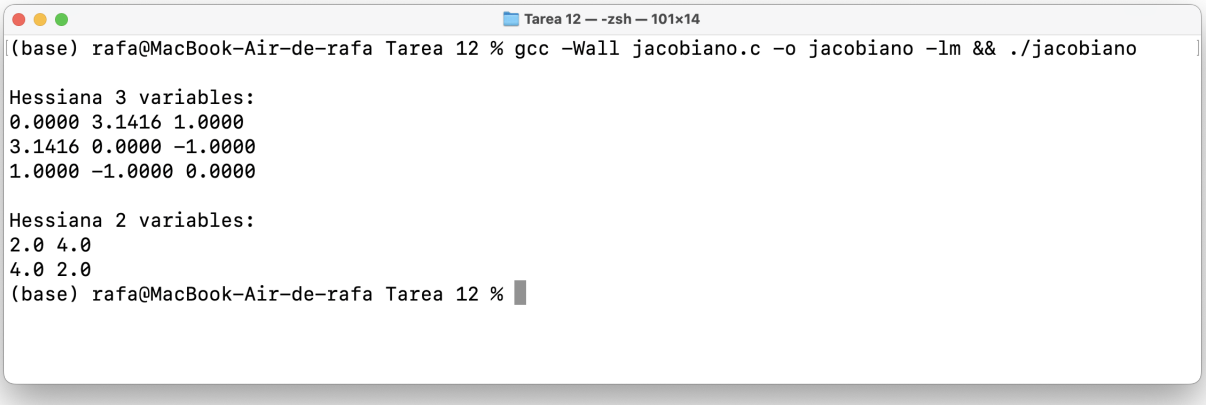
Tenemos los siguientes matrices hessianas para $x = 1 = y$

$$H_f = \begin{pmatrix} 2 & 4 \\ 4 & 2 \end{pmatrix}$$

Para tres variables con $x = 0$, $y = 1$ y $z = \pi$ tenemos

$$H_g = \begin{pmatrix} 0 & \pi & 1 \\ \pi & 0 & -1 \\ 1 & -1 & 0 \end{pmatrix}$$

Con $h = 0.001$ tenemos los siguientes resultados



Por lo que podemos confirmar los resultados del algoritmo.

3. El código para el método fijo es le siguiente

```c
#include <stdio.h>

void punto_fijo_nolineal(int N, double *x0, double tolerancia, int iteraciones, double (*f1)(
    double, double), double (*f2)(double, double)) {
    int i_convergencia = 0;
    double x1_anterior, x2_anterior;
    while (iteraciones--) {
        for (int j = 0; j < N; j++) {
            x1_anterior = x0[0];
            x2_anterior = x0[1];
            x0[0] = f1(x0[0], x0[1]);
            x0[1] = f2(x0[0], x0[1]);
        }
        i_convergencia++;
        if (fabs(x0[0] - x1_anterior) < tolerancia && fabs(x0[1] - x2_anterior) < tolerancia) {
            printf("Punto Fijo no lineal: convergencia alcanzada en %d iteraciones\n",
    i_convergencia);
            return;
        }
    }
    printf("Punto Fijo no lineal: NO converge en %d iteraciones\n", i_convergencia);
}
```

4. El código para el método de Newton es el siguiente

```c
#include <stdio.h>
#include <stdlib.h>

```

```c
void newton_nolineal(int N, double **J, double *F, double *x0, double tolerancia, double h, int iteraciones, double (*f1)(double,double), double (*f2)(double,double)){
    int i_convergencia = 0;
    double *x_anterior = (double *) malloc(N * sizeof(double));
    double dummy;
    while(iteraciones--){
        F[0] = f1(x0[0], x0[1]);
        F[1] = f2(x0[0], x0[1]);
        jacobiano2(J, x0[0], x0[1], h, f1, f2);

        dummy = -J[1][0] / J[0][0];
        J[1][1] -= dummy * J[0][1];
        F[1] -= dummy * F[0];

        x_anterior[1] = F[1] / J[1][1];
        x_anterior[0] = (F[0] - J[0][1] * x_anterior[1]) / J[0][0];

        x0[0] += x_anterior[0];
        x0[1] += x_anterior[1];

        if(fabs(x_anterior[0]) < tolerancia && fabs(x_anterior[1]) < tolerancia){
            printf("Newton no lineal: convergencia alcanzada en %d iteraciones\n", i_convergencia);
            free(x_anterior);
            return;
        }
        i_convergencia++;
    }
    printf("Newton no lineal: no converge en %d iteraciones\n", i_convergencia);
    free(x_anterior);
}
```

5. El código para el método de Broyden es el siguiente

```c
#include <stdio.h>
#include <math.h>

void broyden(double *solucion, double tolerancia, int iteraciones, double (*f1)(double, double), double (*f2)(double, double)) {
    int iteracion = 0;
    double F[2], s[2], u[2], v_anterior[2], w_anterior[2], yk[2], z[2];
    double matriz_jacobiana[2][2];
    double producto_punto;

    matriz_jacobiana[0][0] = 1.0, matriz_jacobiana[0][1] = 1.0;
    matriz_jacobiana[1][0] = 2.0, matriz_jacobiana[1][1] = 2.0;

    while (iteraciones--) {
        w_anterior[0] = v_anterior[0];
        w_anterior[1] = v_anterior[1];
        F[0] = f1(solucion[0], solucion[1]);
        F[1] = f2(solucion[0], solucion[1]);
        v_anterior[0] = F[0];
        v_anterior[1] = F[1];

        yk[0] = v_anterior[0] - w_anterior[0];
        yk[1] = v_anterior[1] - w_anterior[1];

        z[0] = -matriz_jacobiana[0][0] * yk[0] - matriz_jacobiana[0][1] * yk[1];
        z[1] = -matriz_jacobiana[1][0] * yk[0] - matriz_jacobiana[1][1] * yk[1];

        u[0] = s[0] * matriz_jacobiana[0][0] + s[1] * matriz_jacobiana[1][0];
        u[1] = s[0] * matriz_jacobiana[0][1] + s[1] * matriz_jacobiana[1][1];

        producto_punto = u[0] * z[0] + u[1] * z[1];
        iteracion++;

        if (fabs(producto_punto) < tolerancia) {
            printf("Error Broyden: La matriz no es invertible. El m todo diverge.\n");
            return;
        }

        matriz_jacobiana[0][0] += (s[0] + z[0]) * u[0] / producto_punto;
        matriz_jacobiana[0][1] += (s[0] + z[0]) * u[1] / producto_punto;
        matriz_jacobiana[1][0] += (s[1] + z[1]) * u[0] / producto_punto;
        matriz_jacobiana[1][1] += (s[1] + z[1]) * u[1] / producto_punto;

        s[0] = -matriz_jacobiana[0][0] * F[0] - matriz_jacobiana[0][1] * F[1];
        s[1] = -matriz_jacobiana[1][0] * F[0] - matriz_jacobiana[1][1] * F[1];
```

```
        solucion[0] += s[0];
        solucion[1] += s[1];

        if (fabs(s[0]) < tolerancia && fabs(s[1]) < tolerancia) {
            printf("M todo Broyden: Convergencia alcanzada en %d iteraciones\n", iteracion);
            return;
        }
    }
    printf("Error Broyden: El m todo no converge despu s de %d iteraciones.\n", iteracion);
}
```

6. El código para el método de método del gradiente conjugado de Fletcher-Reeves es el siguiente

```
#include <stdio.h>
#include <math.h>

void gradiente_conjugado_nolineal(double *solucion, double tolerancia, int iteraciones, double (*
    f1)(double, double), double (*f2)(double, double)) {
    int iteracion = 0;
    double F[2], direccion[2], gradiente[2];
    double gradiente_anterior[2], direccion_anterior[2];
    double alpha, beta;

    while (iteraciones--) {
        F[0] = f1(solucion[0], solucion[1]);
        F[1] = f2(solucion[0], solucion[1]);
        gradiente[0] = F[0], gradiente[1] = F[1];

        if (iteracion == 0) {
            direccion[0] = -gradiente[0];
            direccion[1] = -gradiente[1];
        } else {
            beta = (gradiente[0] * gradiente[0] + gradiente[1] * gradiente[1]) /
                   (gradiente_anterior[0] * gradiente_anterior[0] + gradiente_anterior[1] *
    gradiente_anterior[1]);
            direccion[0] = -gradiente[0] + beta * direccion_anterior[0];
            direccion[1] = -gradiente[1] + beta * direccion_anterior[1];
        }
        alpha = 0.3;

        solucion[0] += alpha * direccion[0];
        solucion[1] += alpha * direccion[1];

        iteracion++;
        if (fabs(gradiente[0]) < tolerancia && fabs(gradiente[1]) < tolerancia) {
            printf("Gradiente Conjugado NoLineal: Convergencia alcanzada en %d iteraciones\n",
    iteracion);
            return;
        }
        gradiente_anterior[0] = gradiente[0], gradiente_anterior[1] = gradiente[1];
        direccion_anterior[0] = direccion[0], direccion_anterior[1] = direccion[1];
    }
    printf("Error Gradiente Conjugado NoLineal: No convergi  despu s de %d iteraciones",
    iteracion);
}
```

7. a) Para el primer sistema se obtuvieron los siguientes valores para una tolerancia de 1×10^{-10}

Método	Soluciones
Iteración punto fijo	[0.0,3.0]
Newton	[0.0, 3.0]
Broyden	[0.0,3.0]
Gradiente Conjugado	[0.0,3.0]

b) Para el segundo sistema de ecuaciones se obtuvieron lo siguientes resultados para una tolerancia de 1×10^{-10}

Método	Soluciones
Iteración punto fijo	[0.5, 0.014892, -0.1]
Newton	[0.5, 0.014871, -0.1]
Broyden	[0.5, 0.014891, -0.1]
Gradiente Conjugado	NaN

4. Conclusiones

En lo general los todos lo métodos convergen de manera adecuada y sin problemas, sin embargo hemos de notar que, en el caso del sistema de ecuaciones no lineales, lo métodos convergen a una de las soluciones posibles, por lo que no son capaces de brindarlos toda la información posible acerca del sistema, lo cuál no nos asegura proporcionarnos información acerca de la convergencia real del sistema. Esto es, que si un sistema tiene una solución pero estos son incapaces de llegar a ello.

Por otra parte, también vemos que los métodos de diferenciación para datos nos proporciona información limitada acerca del modelo real, pues se limita solamente a realizar aquellos cálculos posibles, siendo que si hace falta más información se tenga que recurrir a algún método de interpolación.

5. Referencias

1. Arévalo Ovalle, D., Bernal Yermanos, M. Á., & Posada Restrepo, J. A. (2021). Métodos numéricos con Python.
2. Kong, Q., Siauw, T., & Bayen, A. (2020). Python programming and numerical methods: A guide for engineers and scientists. Academic Press.
3. Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Editorial Thomson Learning, 2002.
4. Samuel S M Wong, Computational Methods in Physics and Enginering, Ed. World Scientific, 3rd Edition, 1997.
5. Teukolsky, S. A., Flannery, B. P., Press, W. H., & Vetterling, W. T. (1992). Numerical recipes in C. SMR, 693(1), 59-70.
6. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007