TAREA 12 - METODOS NUMERICOS

Guillermo Segura Gómez
Centro de Investigación en Matemáticas
Métodos Numéricos
12 de noviembre de 2023

1 Introducción

La derivada es una operación fundamental en las ciencias. Se utiliza ampliamente en las matemáticas y la física. Pueden ser desde razones de cambio, hasta funciones elementales en el análisis matemático. Tiene importante interés en la física, por ejemplo la derivada de la velocidad de un autobús con respecto al tiempo representa su velocidad. Resolver las derivadas numéricamente se vuelve un problema interesante cuando tratamos con funciones complejas. Existen varias técnicas de derivación que exploran este resultado. Estas técnicas, que incluyen la derivación hacia adelante, hacia atrás, centrada, y los métodos de 3 y 5 puntos, son pilares fundamentales en el estudio de la variación de funciones. Cada una de estas técnicas ofrece una aproximación única a la primera derivada, permitiéndonos explorar las sutilezas del cambio y la tasa de variación en diferentes contextos.

Por otro lado, la solución de sistemas no lineales se presenta como un desafío súper interesante a resolver. Los sistemas no lineales son bastante comunes en la física, por ejemplo algunas de las ecuaciones que solucionan un sistema de presiones, son soluciones de ecuaciones diferenciales ordinarias, que son exponenciales que se acoplan en sistemas de ecuaciones [1]. Resolver este tipo de sistemas físicos se vuelve una tarea ardúa, por lo que es bastante conveniente buscar souciones numéricas. Los métodos de punto fijo, Newton, Broyden y Fletcher-Reeves son técnicas sofisticadas que permiten abordar estos sistemas con eficacia. Cada uno de estos métodos ofrece un enfoque distinto y poderoso para encontrar las raíces de ecuaciones no lineales.

2 Problema 1

Programar en C las técnicas de derivación para la primera (derivación hacia adelante, hacia atrás, centrada, 3 puntos, 5 puntos vistos en clase) y segunda derivada (fórmula del punto medio visto en clase) de una función arbitraria. Utilizando la siguiente tabla y las fórmulas programadas, aproxima adecuadamente $f'(1,3)$ y $f''(1,3)$, con h =0.1, 0.01, según corresponda y compara con los verdaderos valores si $f(x) = 3xe^x - \cos(x)$.

2.1 Pseudocódigo

Se presenta el pseudocódigo para una rutina que calcula la derivada de una función en un punto. Las funciones de este problema son similares por lo que solo se presenta un pseudocódigo. Las rutinas para calcular las derivadas se encuentran en la librería **derivation**.

```
function FivePointDerivative(f: Function(x: Double): Double; x, h: Double):
    Double;
begin
    FivePointDerivative := (f(x - 2 * h) - 8 * f(x - h) + 8 * f(x + h) - f(x
    + 2 * h)) / (12 * h);
end;
```

Listing 1: Derivación de cinco puntos con O(h4)

2.2 Ejecución

Ahora se presenta la ejecución para diferentes valores de x en los que se muestra una tabla comparativa con los resultados.

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/DerivationExamples.o -c DerivationExamples.c
gcc -g -o build/DerivationExamples build/Debug/derivation.o build/Debug/
    matrix.o build/Debug/DerivationExamples.o
guillermo_sego@MacBook-Air Tarea12 % ./build/DerivationExamples
Comparación de derivada evaluada en x = 1.200000
| Método          | h=0.1            | h=0.01           | Valor verdadero |
----------------------------------------------------------------------
| Hacia adelante  | 24.5269500772    | 23.0066720159    | 22.8448107760    |
| Hacia atrás    | 21.2991140978    | 22.6843129278    | 22.8448107760    |
| Centrada        | 22.9130320875    | 22.8454924718    | 22.8448107760    |
| Tres Puntos     | 22.6948524019    | 22.8434344636    | 22.8448107760    |
| Cinco Puntos    | 22.8446015013    | 22.8448107551    | 22.8448107760    |
| Segunda Deriv.  | 32.2783597938    | 32.2359088103    | 32.2354802127    |
```

Listing 2: Ejecución x

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/DerivationExamples.o -c DerivationExamples.c
gcc -g -o build/DerivationExamples build/Debug/derivation.o build/Debug/
    matrix.o build/Debug/DerivationExamples.o
guillermo_sego@MacBook-Air Tarea12 % ./build/DerivationExamples
Comparación de derivada evaluada en x = 1.300000
| Método          | h=0.1            | h=0.01           | Valor verdadero |
----------------------------------------------------------------------
| Hacia adelante  | 28.1911454276    | 26.4654481363    | 26.2817051920    |
| Hacia atrás    | 24.5269500772    | 26.0995079384    | 26.2817051920    |
| Centrada        | 26.3590477524    | 26.2824780373    | 26.2817051920    |
| Tres Puntos     | 26.1117636563    | 26.2801449121    | 26.2817051920    |
| Cinco Puntos    | 26.2814704554    | 26.2817051686    | 26.2817051920    |
| Segunda Deriv.  | 36.6419535041    | 36.5940197929    | 36.5935358381    |
```

Listing 3: Ejecución x

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/DerivationExamples.o -c DerivationExamples.c
gcc -g -o build/DerivationExamples build/Debug/derivation.o build/Debug/
    matrix.o build/Debug/DerivationExamples.o
guillermo_sego@MacBook-Air Tarea12 % ./build/DerivationExamples
Comparación de derivada evaluada en x = 1.400000
| Método          | h=0.1            | h=0.01           | Valor verdadero |
----------------------------------------------------------------------
| Hacia adelante  | 32.3499089701    | 30.3914329819    | 30.1828894913    |
| Hacia atrás    | 28.1911454276    | 29.9760974535    | 30.1828894913    |
| Centrada        | 30.2705271988    | 30.1837652177    | 30.1828894913    |
| Tres Puntos     | 29.9904057324    | 30.1811215782    | 30.1828894913    |
| Cinco Puntos    | 30.1826262717    | 30.1828894650    | 30.1828894913    |
| Segunda Deriv.  | 41.5876354249    | 41.5335528428    | 41.5330068047    |
```

Listing 4: Ejecución x

3 Problema 2

Con los códigos de derivación programa nuevo código que calcule la matriz Jacobiana y la matriz Hessiana de un sistema de ecuaciones no lineales. Comprueba tus cálculos con los verdaderos valores de una sistema no lineal, puedes basarte en los ejemplos vistos en clase.

3.1 Pseudocódigo

Se presenta el pseudocódigo de la rutina que calcula primero la primera derivada parcial utilizando el método de cinco puntos para implementar la rutina de la matriz jacobiana.

```
function PartialDerivative(i: Integer; var x: array of Double; xi: Integer; h
    : Double): Double;
var
    originalValue, f1, f2, f3, f4: Double;
begin
    originalValue := x[xi];
    x[xi] := originalValue - 2 * h;
    f1 := SystemFunctions[i](x);
    x[xi] := originalValue - h;
    f2 := SystemFunctions[i](x);
    x[xi] := originalValue + h;
    f3 := SystemFunctions[i](x);
    x[xi] := originalValue + 2 * h;
    f4 := SystemFunctions[i](x);
    x[xi] := originalValue;
    PartialDerivative := (f1 - 8 * f2 + 8 * f3 - f4) / (12 * h);
end;
```

Listing 5: Rutina para calcular la primera derivada parcial

```
procedure Jacobian(var x: array of Double; var jacobiano: array of array of
    Double; h: Double; N: Integer);
var
    i, j: Integer;
begin
    for i := 0 to N - 1 do
        for j := 0 to N - 1 do
            jacobiano[i][j] := PartialDerivative(i, x, j, h);
end;
```

Listing 6: Rutina para calcular la matriz jacobiana

Se presenta ahora el pseudocódigo para la segunda derivada parcial, con la cual se calcula la matriz hessiana.

```
function SecondPartialDerivative(funcIndex: Integer; var x: array of Double;
    xi, xj: Integer; h: Double): Double;
var
    originalValueXi, originalValueXj, f1, f2, f3, f4: Double;
begin
    originalValueXi := x[xi];
    originalValueXj := x[xj];
    x[xi] := originalValueXi + h; x[xj] := originalValueXj + h;
    f1 := SystemFunctions[funcIndex](x);
    x[xi] := originalValueXi - h; x[xj] := originalValueXj + h;
    f2 := SystemFunctions[funcIndex](x);
```

```
    x[xi] := originalValueXi + h; x[xj] := originalValueXj - h;
    f3 := SystemFunctions[funcIndex](x);
    x[xi] := originalValueXi - h; x[xj] := originalValueXj - h;
    f4 := SystemFunctions[funcIndex](x);
    x[xi] := originalValueXi; x[xj] := originalValueXj;
    SecondPartialDerivative := (f1 - f2 - f3 + f4) / (4 * h * h);
end;
```

Listing 7: Rutina para calcular la segunda derivada parcial

```
procedure Hessian(var x: array of Double; var hessiana: array of array of
    Double; h: Double; N: Integer);
var
    i, j: Integer;
begin
    for i := 0 to N - 1 do
        for j := 0 to N - 1 do
            hessiana[i][j] := SecondPartialDerivative(i, x, i, j, h);
end;
```

Listing 8: Rutina para calcular la matriz hessiana

3.2 Ejecución

Los resultados de la ejecución de las matrices se muestra con el siguiente ejemplo de un sistema no lineal del libro Burden [1]

$$3x_1 - \cos(x_2x_3) - 1/2 = 0$$

$$x_1^2 - 81(x_2 + 0.1)^2 + \sin(x_3) + 1.06 = 0$$

$$e^{-x_1x_2} + 20x_3 + \frac{10\pi - 3}{3} = 0$$

Se calcula la matriz jacobiana y hessiana para el sistema

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/Jacobian_Hessian.o -c Jacobian_Hessian.c
gcc -g -o build/Jacobian_Hessian build/Debug/derivation.o build/Debug/matrix.
    o build/Debug/Jacobian_Hessian.o
guillermo_sego@MacBook-Air Tarea12 % ./build/Jacobian_Hessian
Matriz Jacobiana:
3.000000        0.001000        -0.001000
0.200000        -32.400000      0.995004
-0.099005       -0.099005       20.000000
Matriz Hessiana:
0.000000        0.000000        0.000000
0.000000        0.000000        0.000000
-0.000004       -0.000004       0.000000
```

Listing 9: Ejecución

4 Problema 3

Programar en C el método de iteración de punto fijo para sistema de ecuaciones no lineales general.

4.1 Pseudocódigo

Se presenta el pseudocódigo para la función de punto fijo a continuación

```
procedure FixedPointIteration(var x: array of Double; tol: Double; maxIter, N
    : Integer);
var
    xNext: array of Double;
    error: Double;
    iter, i: Integer;
begin
    SetLength(xNext, N);
    iter := 0;

    repeat
        // Aplicar la función G
        G(x, xNext);

        // Calcular el error como la norma de la diferencia entre iteraciones
        for i := 0 to N - 1 do
            xNext[i] := xNext[i] - x[i];
        error := Norm(xNext, N);

        // Preparar la siguiente iteración
        for i := 0 to N - 1 do
            x[i] := x[i] + xNext[i];

        iter := iter + 1;
    until (error <= tol) or (iter >= maxIter);

    // Imprimir el resultado
    if iter < maxIter then
    begin
        WriteLn('Solución encontrada después de ', iter, ' iteraciones:');
        for i := 0 to N - 1 do
            WriteLn('x[', i, '] = ', x[i]:0:6);
    end
    else
        WriteLn('No se encontró una solución en ', maxIter, ' iteraciones.');
end;
```

Listing 10: Rutina para calcular el método de punto fijo

4.2 Ejecución

Se presenta la ejecución del método de punto fijo

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/fixedPoint.o -c fixedPoint.c
gcc -g -o build/fixedPoint build/Debug/derivation.o build/Debug/matrix.o
    build/Debug/fixedPoint.o
guillermo_sego@MacBook-Air Tarea12 % ./build/fixexPoint
zsh: no such file or directory: ./build/fixexPoint
guillermo_sego@MacBook-Air Tarea12 % ./build/fixedPoint
Solución encontrada después de 5 iteraciones:
x[0] = 0.500000
x[1] = 0.000000
```

```
x[2] = -0.523599
```

Listing 11: Ejecución

5 Problema 4

Programar en C el método de iteración de Newton para sistema de ecuaciones no lineales general.

5.1 Pseudocódigo

Se presenta el pseudocódigo para la función de Newton a continuación

```
procedure NewtonMethod(var x: array of Double; tol: Double; maxIter, N:
    Integer);
var
    Fval, deltaX: array of Double;
    h: Double;
    iter, i: Integer;
    Jval: array of array of Double;
    J_flat: array of Double;
begin
    SetLength(Fval, N);
    SetLength(deltaX, N);
    h := 1e-5;  // Paso para el cálculo de derivadas

    // Crear Jval como una matriz dinámica
    SetLength(Jval, N, N);

    for iter := 0 to maxIter - 1 do
    begin
        F(x, Fval, N); // Evaluar la función
        Jacobian(x, Jval, h, N); // Calcular el Jacobiano

        // Convertir el Jacobiano y Fval para usar en el método del Gradiente
     Conjugado
        J_flat := FlattenMatrix(Jval, N, N);
        NegateVector(Fval, N); // Convertir a -Fval

        // Resuelve el sistema lineal J * deltaX = -F usando Gradiente
    Conjugado
        ConjugateGradient(J_flat, Fval, deltaX, N, N);

        // Actualizar x = x + deltaX
        for i := 0 to N - 1 do
            x[i] := x[i] + deltaX[i];

        // Verificar la convergencia
        if Norm(deltaX, N) < tol then
            break;

        // Liberar la memoria de la matriz aplanada
        // (En Pascal, no es necesario liberar memoria dinámica manualmente)
    end;
```

```
    // Comprobar si se alcanzó la convergencia
    if iter = maxIter then
        WriteLn('No se alcanzó la convergencia después de ', maxIter, '
    iteraciones.')
    else
        WriteLn('Convergencia alcanzada en ', iter, ' iteraciones.');
end;
```

Listing 12: Rutina para calcular el método de Newton

5.2 Ejecución

Se presenta la ejecución del método de Newton

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/NewtonNoLineal.o -c NewtonNoLineal.c
gcc -g -o build/NewtonNoLineal build/Debug/derivation.o build/Debug/matrix.o
    build/Debug/NewtonNoLineal.o
guillermo_sego@MacBook-Air Tarea12 % ./build/NewtonNoLineal
Convergencia alcanzada en 23 iteraciones.
Solución encontrada:
0.499975
-0.000003
-0.523597
```

Listing 13: Ejecución

6 Problema 5

Programar en C el método de iteración de Broyden para sistema de ecuaciones no lineales general.

6.1 Pseudocódigo

Se presenta el pseudocódigo para la función de Broyden a continuación. Para la implementación del método de Broyden se utilizó una rutina auxiliar cuya función era actualizar la matriz tras cada iteración. Este método exige calcular la inversa de una matriz A, para implementar esto, se utilizó el método de gauss - jordan, en el que se implementaba sobre una matriz aumentada. Se muestran los pseudocódigos del método de Broyden y de la función auxiliar.

```
procedure UpdateBroyden(var A: array of Double; var s, y, v: array of Double;
     n: Integer);
var
    z, u: array of Double;
    p, temp: Double;
    i, j: Integer;
begin
    SetLength(z, n);
    SetLength(u, n);

    // Paso 6: z = -A * y
    MultiplyArrayWithScalar(y, -1, y, n);
```

```
    MatrixProduct(A, y, z, n, n, 1);

    // Paso 7: p = -s^T * z
    p := 0;
    for i := 0 to n - 1 do
        p := p - s[i] * z[i];

    // Paso 8: u^T = s^T * A
    for i := 0 to n - 1 do
    begin
        u[i] := 0;
        for j := 0 to n - 1 do
            u[i] := u[i] + s[j] * A[j * n + i];
    end;

    // Paso 9: A = A + (1/p) * (s + z) * u^T
    for i := 0 to n - 1 do
        for j := 0 to n - 1 do
            A[i * n + j] := A[i * n + j] + (1 / p) * (s[i] + z[i]) * u[j];
end;
```

Listing 14: Rutina auxiliar

```
procedure BroydenMethod(var x: array of Double; tol: Double; maxIter, n:
    Integer);
var
    deltaX, v, w, y, Ainv: array of Double;
    A: array of array of Double;
    iter, i: Integer;
begin
    SetLength(deltaX, n);
    SetLength(A, n, n);
    SetLength(v, n);
    SetLength(w, n);
    SetLength(y, n);

    // Paso 1: Determinar A_0 = J(x) y v = F(x)
    Jacobian(x, A, tol, n);
    F(x, v, n);

    // Paso 2: A = A_0^-1 (Usar eliminación gaussiana)
    GaussJordanInverse(n, A, A);

    // Aplanamos la matriz
    Ainv := FlattenMatrix(A, n, n);

    for iter := 1 to maxIter do
    begin
        // Paso 3: s = -A * v
        MultiplyArrayWithScalar(v, -1, v, n);
        MatrixProduct(Ainv, v, deltaX, n, n, 1);

        // Paso 4: x = x + s
        for i := 0 to n - 1 do
            x[i] := x[i] + deltaX[i];

        // Paso 5: w = v; v = F(x); y = v - w
```

```
        CopyArray(v, w, n);
        F(x, v, n);
        for i := 0 to n - 1 do
            y[i] := v[i] - w[i];

        // Actualizar A usando Broyden
        UpdateBroyden(Ainv, deltaX, y, v, n);

        // Verificar la convergencia
        if Norm(deltaX, n) < tol then
        begin
            WriteLn('Convergencia alcanzada en ', iter, ' iteraciones.');
            Break;
        end;
    end;

    if iter > maxIter then
        WriteLn('Número máximo de iteraciones excedido.');
end;
```

Listing 15: Rutina para calcular el método de Broyden

6.2 Ejecución

Se presenta la ejecución del método de Broyden

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/Broyden.o -c Broyden.c
gcc -g -o build/Broyden build/Debug/derivation.o build/Debug/matrix.o build/
    Debug/Broyden.o
guillermo_sego@MacBook-Air Tarea12 % ./build/Broyden
Convergencia alcanzada en 14 iteraciones.
Solución encontrada:
0.495976
0.012451
-0.518585
```

Listing 16: Ejecución

7 Problema 5

Programar en C el método de iteración de Fletcher-Reevers para sistema de ecuaciones no lineales general.

7.1 Pseudocódigo

Se presenta el pseudocódigo para la función de Fletcher-Reevers a continuación. Este método exige el cálculo del gradiente de la función. Debido a que se trabaja con un polinomio cuadrático el gradiente pasa a tener la siguiente forma

$$\nabla g = 2J(x)^T F(x)$$

donde J(x) es la matriz jacobiana.

```pascal
procedure FletcherReeves(var x: array of Double; n: Integer; tol: Double;
    maxIter: Integer);
var
    g1, g2, g3, g0, g_new, alpha0, alpha1, alpha2, alpha3, z0, h1, h2, h3:
    Double;
    z, x_new, F_val, JacobianFlat: array of Double;
    JacobianMatrix: array of array of Double;
    k, i: Integer;
    alpha: Double;
begin
    SetLength(z, n);
    SetLength(x_new, n);
    SetLength(F_val, 3); // Suponiendo que F_val tiene 3 elementos
    SetLength(JacobianMatrix, 3); // Suponiendo 3 filas para la matriz
    Jacobiana
    for i := 0 to 2 do
        SetLength(JacobianMatrix[i], n);
    SetLength(JacobianFlat, n * 3);

    alpha1 := 0;
    alpha3 := 1;

    for k := 1 to maxIter do
    begin
        F(x, F_val, n); // Calcula F(x)
        Jacobian(x, JacobianMatrix, tol, n); // Calcula la matriz Jacobiana

        // Aplana la matriz Jacobiana
        FlattenMatrix(JacobianMatrix, JacobianFlat, n, n);

        // Calcula el gradiente como 2 * Jacobian^T * F_val
        MatrixTranspose(n, n, JacobianFlat, JacobianFlat); // Transpone la
    Jacobiana
        MatrixProduct(JacobianFlat, F_val, z, n, 3, 1); // Multiplica
    Jacobian^T * F_val
        MultiplyArrayWithScalar(z, 0.5, z, n); // Multiplica por 2

        z0 := Norm(z, n);
        if z0 = 0 then
        begin
            WriteLn('Gradiente cero en la iteración ', k);
            Break;
        end;

        // Normalizar z
        DivideArrayWithScalar(z, z0, z, n);

        g1 := g(x, n);
        while True do
        begin
            for i := 0 to n - 1 do
                x_new[i] := x[i] - alpha3 * z[i];
            g3 := g(x_new, n);

            if g3 < g1 then
                Break;
```

```
            alpha3 := alpha3 / 2;
            if alpha3 < tol then
            begin
                WriteLn('Sin probable mejora en la iteración ', k);
                Break;
            end;
        end;

        alpha2 := alpha3 / 2;
        for i := 0 to n - 1 do
            x_new[i] := x[i] - alpha2 * z[i];
        g2 := g(x_new, n);

        // Cálculos para interpolar la cuadrática
        h1 := (g2 - g1) / alpha2;
        h2 := (g3 - g2) / (alpha3 - alpha2);
        h3 := (h2 - h1) / alpha3;
        alpha0 := 0.5 * (alpha2 - h1 / h3);

        for i := 0 to n - 1 do
            x_new[i] := x[i] - alpha0 * z[i];
        g0 := g(x_new, n);

        // Escoger el mejor alpha
        if g0 < g3 then
            alpha := alpha0
        else
            alpha := alpha3;
        for i := 0 to n - 1 do
            x[i] := x[i] - alpha * z[i];
        g_new := g(x, n);

        // Verificar la convergencia
        if Abs(g_new - g1) < tol then
        begin
            WriteLn('Convergencia alcanzada en la iteración ', k);
            Break;
        end;
    end;

    if k = maxIter then
        WriteLn('Número máximo de iteraciones excedido');
end;
```

Listing 17: Rutina para calcular el método de Fletcher-Reevers

7.2 Ejecución

Se presenta la ejecución del método de Fletcher-Reevers

```
guillermo_sego@MacBook-Air Tarea12 % make
gcc -g -o build/Debug/CG_Fletcher.o -c CG_Fletcher.c
gcc -g -o build/CG_Fletcher build/Debug/derivation.o build/Debug/matrix.o
    build/Debug/CG_Fletcher.o
guillermo_sego@MacBook-Air Tarea12 % ./build/CG_Fletcher
Convergencia alcanzada en la iteración 60
Solución encontrada:
```

```
0.497142
-0.000180
-0.523492
```

Listing 18: Ejecución

8 Conclusión

En este trabajo, hemos profundizado en las técnicas de derivación y la solución de sistemas no lineales, dos áreas fundamentales en el campo del análisis numérico. Las técnicas de derivación, incluyendo la derivada hacia adelante, hacia atrás, centrada, y los métodos de 3 y 5 puntos, han demostrado ser herramientas esenciales para entender y aproximar las variaciones de funciones complejas.

Por otro lado, la resolución de sistemas no lineales, a través de métodos como punto fijo, Newton, Broyden y Fletcher-Reeves, revela la complejidad y la belleza inherente en la búsqueda de soluciones a ecuaciones que modelan fenómenos del mundo real.

En conclusión, este estudio no solo subraya la importancia y la utilidad de estas técnicas de derivación y resolución de sistemas no lineales, sino que también ilustra la interconexión entre teoría y práctica en el análisis numérico. Así como en trabajos anteriores donde exploramos la aproximación de mínimos cuadrados y la integración numérica, este trabajo destaca la relevancia de seleccionar la herramienta adecuada para cada problema específico, balanceando precisión y practicidad para obtener resultados óptimos en el mundo de las ciencias y la ingeniería.

References

[1] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis.* Cengage learning, 2015.