FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP1 Modelacion Numerica

Grupo Nº	**4**
Salvador Perez Mendoza	**110198**
Juan Abdala	**112154**
Lucas Araujo	**109867**

Fecha	**Correcciones**	**Docente**

Calificación Final	Docente	Fecha

Índice

1. Introducción

En este trabajo práctico se abordarán los métodos iterativos de Jacobi y Gauss-Seidel, dos técnicas fundamentales en la resolución de sistemas de ecuaciones lineales de la forma Ax=b

Estos métodos son especialmente útiles cuando se trabaja con matrices grandes, donde los métodos directos, como la eliminación gaussiana, pueden ser ineficientes o inviables debido a su alto costo computacional.

El método de Jacobi y el de Gauss-Seidel son iterativos, lo que significa que a partir de una suposición inicial se refina progresivamente la solución mediante la repetición de un conjunto de operaciones hasta alcanzar un nivel de convergencia aceptable. Ambos métodos difieren en la forma en que actualizan las aproximaciones a las incógnitas en cada iteración.

El objetivo de este trabajo es implementar, en el lenguaje de programacion Octave, ambos algoritmos y analizarlos, examinando su convergencia y comportamiento en diferentes escenarios. Para esto, se resolverán varios sistemas de ecuaciones con distintas características, permitiendo comparar la eficiencia y el rendimiento de cada método bajo condiciones diversas.

2. a) Armado del sistema de ecuaciones lineales

Para Armar el sistema se uso la Matriz Ax=b tal que:

$$\left|\begin{array}{cccccccc} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{array}\right| \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_n \end{bmatrix}$$

Figura 1: Sistema de ecuaciones lineaes que vamos a analizar

Donde se obtuvo el a partir de la formula:

$$\rho_j = h^2 \cdot \left(-x_j \cdot \left(x_j + 3\right) \cdot e^{x_j}\right) \quad j = 2,3,\dots,n$$

Figura 2: Formula para

Y el h se obtubo a partir de la formula:

$$h=1/(n-1) \text{ y } x_j = h * (j-1).$$

Figura 3: formula para h

Como se puede notar la inicializacion del sistema no se hace de manera aleatoria sino que depende el n que se le pase, siendo n el tamaño de filas y columnas que va a tener la matriz.

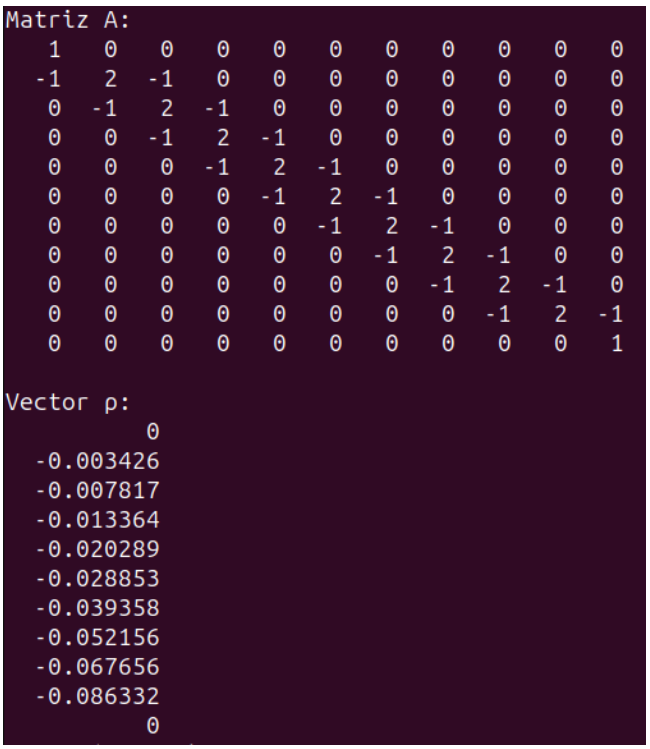

Figura 4: Matriz A y Vector rho generado con un n = 11

3. b) Programacion de los metodos indirectos para sistemas lineales, Jacobi y Gauss-Seidel

3.0.1. Metodo Jacobi

Para programar el metodo de jacobi se utilizo la siguiente formula de Jacobi

$$X_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{ij} X_j^{(k)}}{a_{ii}}$$

El codigo se puede ver en el anexo pero nos basamos en usar for para iterar sobre cada variable del sistema, actualizando su valor en cada paso. Para esto, se utiliza un bucle externo que controla el número de iteraciones, y dentro de este, un bucle interno que recorre las ecuaciones, resolviendo cada una de ellas con base en los valores actuales de las otras variables, siguiendo el esquema de actualización de Gauss-Seidel.

3.0.2. Metodo Gauss Seidel

Para Gauss-Seidel se hizo lo mismo pero aplicando la formula:

$$X_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} X_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} X_j^{(k)}}{a_{ii}}$$

El código se puede ver en el anexo, pero nos basamos en usar un bucle externo para controlar el número de iteraciones, y dentro de este, un bucle interno que itera sobre cada variable del sistema, actualizando su valor en cada paso de manera inmediata. A diferencia del método Jacobi, en Gauss-Seidel cada variable se recalcula utilizando los valores más recientes ya actualizados en la misma iteración, lo que acelera la convergencia en muchos casos. En cada iteración, se resuelve una ecuación para una variable, usando las soluciones actualizadas de las variables anteriores y las antiguas de las siguientes.

4. C) resolver el sistema con n=11 y Tolerancia = 0.00001

Para resolver el problema propuesto utilizando ambos métodos implmentados anteriormente. Se parte de un sistema de tamaño n = 11 donde se emplea una tolerancia de corte TOL = 0,00001. para determinar la precisión

de la solución. El cálculo inicial arranca desde una solución j0 = 1 para cada componente j del sistema (con j = 1,2,...,n). El sistema a resolver es el que se muestra en la Figura 4 Dicha solucion para ambos metodos nos dio:

Claramente dan resultados iguales debido a que nosotros le pusimos la misma tolerancia para ambos metodos. Como se ve el resultado puede cambiar en ambas soluciones en su cuarto decimal ya que tiene un error del 0.00001.

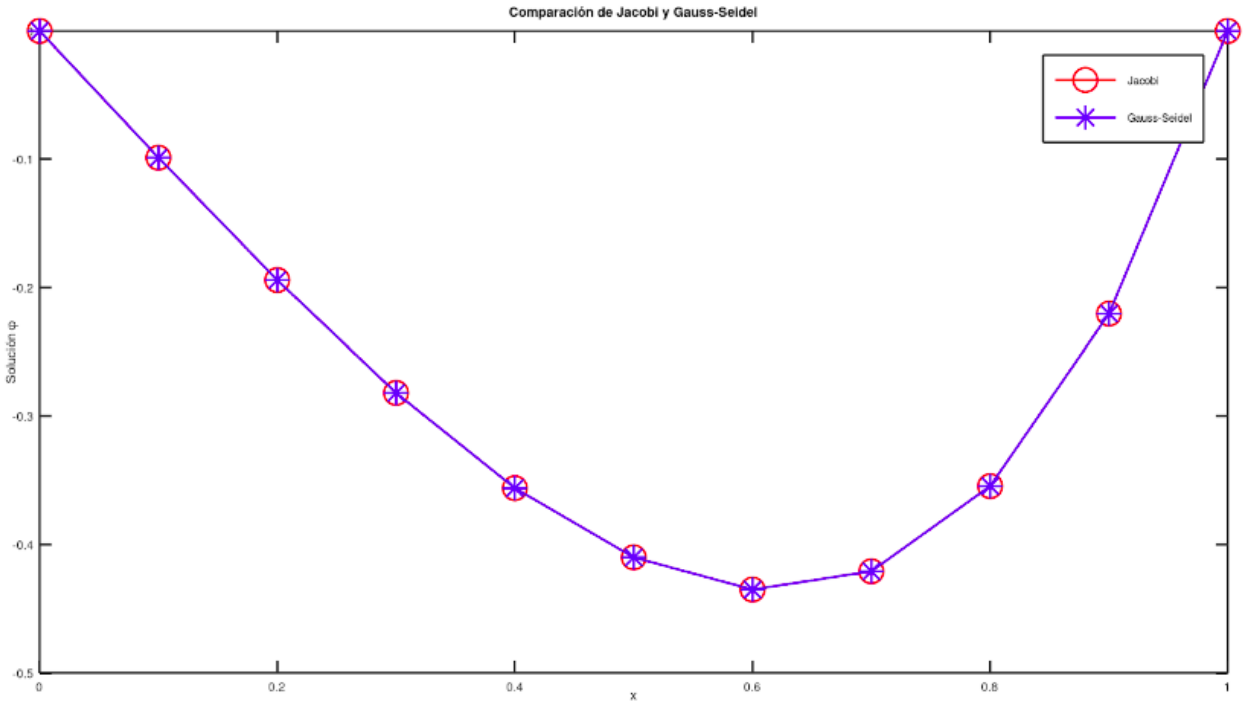

Figura 5: Comparacion de Jacobi y Gauss-Seidel

El grafico empieza con un phi arriba que va bajando y luego sube.

5. D) Analisis de velocidad de convergencia de Jacobi y Gauss-Seidel

Para cada uno de los métodos utilizados en la resolución del sistema para n = 11 con una Tolerancia = 0,00001. Se generara un grafico de convergencia con el objetivo de comparar la eficiencia de ambos métodos en términos de su rapidez para llegar a la solucion.

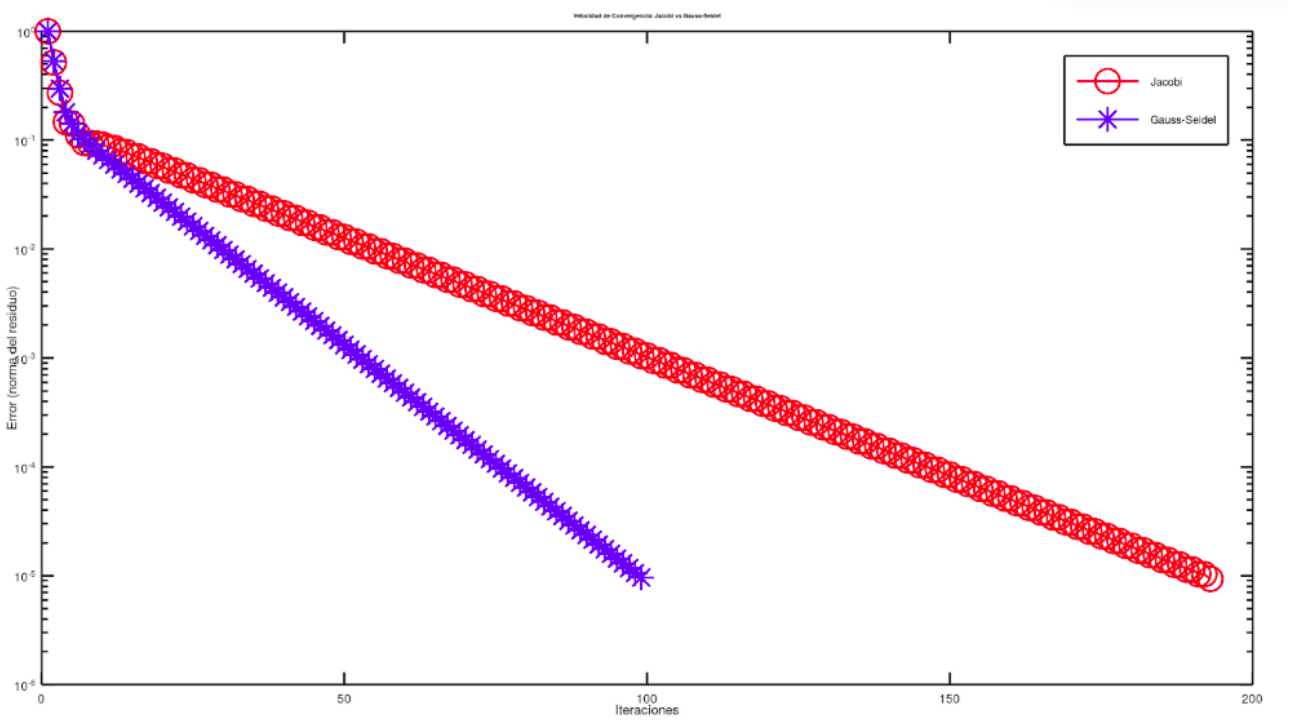

Figura 6: Grafico de convergencia en funcion del tiempo

La gráfica muestra cómo, al iniciar las iteraciones, la norma del error entre la iteración actual y la anterior es considerablemente alta. A medida que se avanza hacia la solución, este error disminuye gradualmente hasta alcanzar el criterio de corte de 0.00001.

Como era de esperar, el método de Jacobi requiere aproximadamente el doble de iteraciones para converger a la misma solución en comparación con el método de Gauss Seidel. Esto se debe a que el método de Gauss Seidel utiliza las incognitas de halladas en la iteracion actual cuando puede, a diferencia de jacobi que usaba siempre las incognitas de de la iteracion anterior. Esto le permite a Gauss Seidel una convergencia más rápida hacia la solución.

La velocidad de convergencia esta dada por las pendientes de las rectas

6. E) Comparación de Performance según el Tamaño del Sistema

# Sistema de tamaño n	# Jacobi (segundos)	# Jacobi (iteraciones)	# Gauss-Seidel (segundos)	# Gauss-Seidel (iteraciones)	Error Jacobi	Error Gauss-Seidel
11	0,09906	193	0,04908	99	2,00E-05	1,00E-05
51	32,94284	3.229	15,01660	1.651	2,00E-05	1,00E-05
101	591,52507	10.112	363,87892	5.197	2,00E-05	1,00E-05

Figura 7: Resultados de performance de Jacobi y Gauss-Seidel mostrados en la consola de OctaveGNU

Para un sistema de tamaño n=11, Jacobi toma 0.099063 segundos y 193 iteraciones, mientras que Gauss-Seidel resuelve el mismo sistema en 0.049076 segundos y 99 iteraciones. La diferencia es considerable, siendo Gauss-Seidel casi el doble de rápido.

En el caso del sistema n=51, Jacobi tarda aproximadamente 32.94 segundos y requiere 3229 iteraciones, mientras que Gauss-Seidel lo resuelve en 15.01 segundos con la mitad de iteraciones (1651). Este patrón de superioridad en rendimiento para Gauss-Seidel se repite de forma clara con n=101, donde Jacobi toma alrededor de 591.52 segundos y 10112 iteraciones, frente a los 363.87 segundos y 5197 iteraciones de Gauss-Seidel.

Por lo tanto, si bien ambos métodos son precisos, Gauss-Seidel ofrece una mejor performance tanto en terminos de tiempo como de iteraciones. Lo cual para problemas de mayor complejidad computacional es mejor optar por el metodo de Gauss-Seidel antes que el de Jacobi

7. F) Comparación de Performance según el Criterio de Corte (Tolerancia)

Tolerancia	# Jacobi (segundos)	# Jacobi (iteraciones)	# Gauss-Seidel (segundos)	# Gauss-Seidel (iteraciones)	Error Jacobi	Error Gauss-Seidel
1,00E-05	474,10458	10.112	222,16877	5.197	2,00E-05	1,00E-05
1,00E-04	252,44413	5.448	132,15829	2.864	2,00E-04	1,00E-04
1,00E-03	31,19712	618	23,61281	500	2,00E-03	1,00E-03

Figura 8: Resultados de performance de Jacobi y Gauss-Seidel mostrados en la consola de OctaveGNU

Para una Tol=0.00001 el método de Jacobi toma 474.1 segundos y 10112 iteraciones, mientras que Gauss-Seidel converge en 222.16 segundos y 5197 iteraciones. La eficiencia de Gauss-Seidel mejora aproximadamente el doble tnanto en iteraciones como la cantidad de segundos que le toma.

Como se puede notar para una Tol=0.001 Jacobi resuelve el sistema en 31.19 segundos y 618 iteraciones, mientras que Gauss-Seidel lo hace en solo 23.61 segundos y 500 iteraciones. En este caso la diferencia entre ambos metodos es menor, igualmente Gauss-Seidel sigue mostrando mayor performance

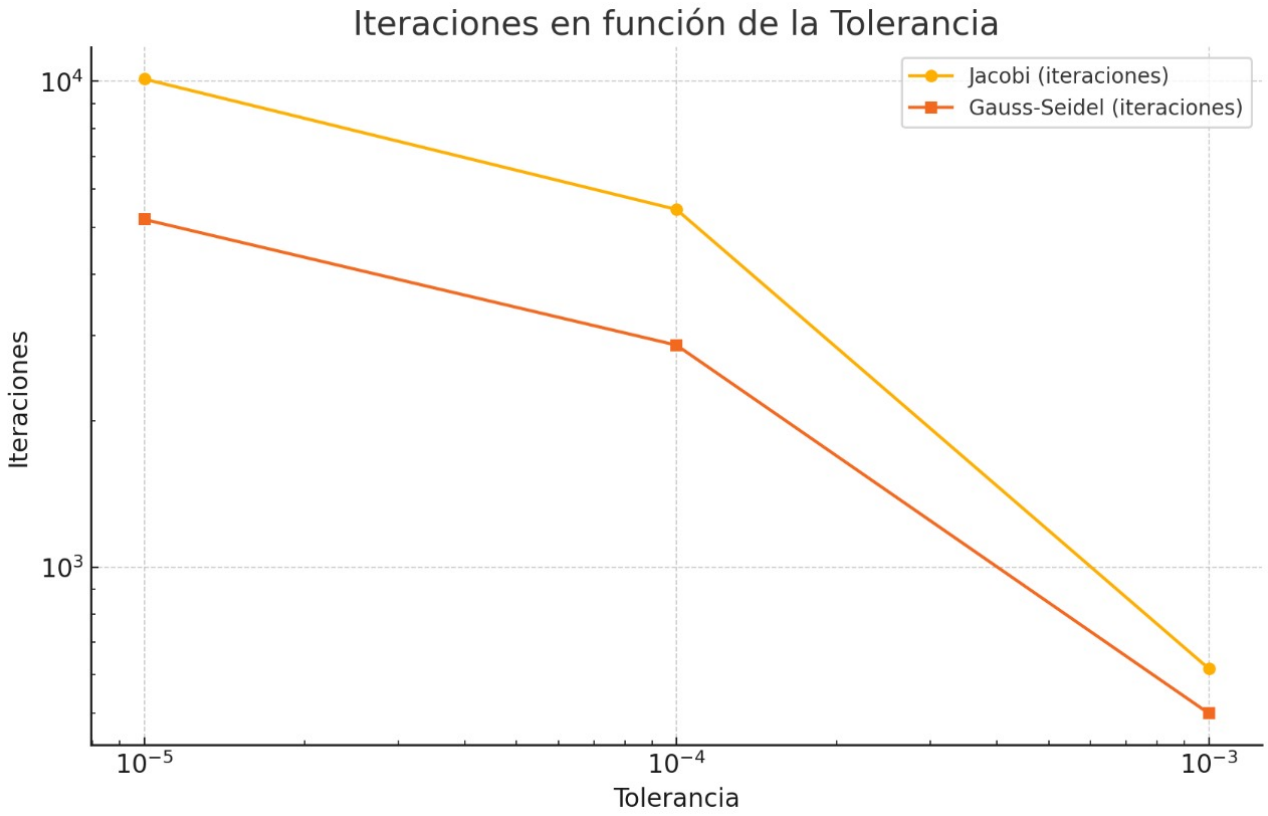

Figura 9: Cantidad de iteraciones en funcion de la tolerancia

El gráfico muestra la cantidad de iteraciones requeridas por los métodos de Jacobi y Gauss-Seidel para alcanzar diferentes niveles de tolerancia en la solución de un sistema de ecuaciones lineales.

8. Conclusiones Finales

A partir de los experimentos realizados con los métodos iterativos Jacobi y Gauss-Seidel para la resolución de sistemas de ecuaciones lineales, se pueden extraer las siguientes conclusiones:

La elección entre ambos métodos depende del problema a resolver. Si se dispone de recursos paralelos significativos, Jacobi puede ser una opción viable. Sin embargo, para la mayoría de los casos, y especialmente en sistemas grandes, Gauss-Seidel es la opción recomendada debido a su mayor eficiencia.

9. Anexo

El codigo esta dividido en 6 archivos/funciones diferentes, basandose principalmente el el main, que es el encargado de tener las variables y comunicarse con el resto para que devuelvan los distintos resultados y luego con estos resultados armar los graficos .

```
function main()
    n = 11;
    tol = 0.00001;
    max_iter = 100000;

    % Generar el sistema
    [A, rho] = generar_sistema(n);

    % Resolver con Jacobi
    [phi_jacobi, error_jacobi] = jacobi(A, rho, tol, max_iter);
    disp("Solución Jacobi:");
    disp(phi_jacobi);

    % Resolver con Gauss-Seidel
    [phi_gauss, error_gauss] = gauss_seidel(A, rho, tol, max_iter);
    disp("Solución Gauss-Seidel:");
    disp(phi_gauss);

      % Gráficos
     x = linspace(0, 1, n);  % Vector de posiciones para graficar
     plot(x, phi_jacobi, 'r-o', 'DisplayName', 'Jacobi');
     hold on;
     plot(x, phi_gauss, 'b-*', 'DisplayName', 'Gauss-Seidel');
     xlabel('x');
     ylabel('Solución \phi');
     title('Comparación de Jacobi y Gauss-Seidel');
     legend show;
     hold off;

    % Gráfico comparativo de la convergencia
    iter_jacobi = length(error_jacobi);
    iter_gauss = length(error_gauss);

    figure;
    semilogy(1:iter_jacobi, error_jacobi, 'r-o', 'DisplayName', 'Jacobi');
    hold on;
    semilogy(1:iter_gauss, error_gauss, 'b-*', 'DisplayName', 'Gauss-Seidel');
    xlabel('Iteraciones');
    ylabel('Error (norma del residuo)');
    title('Velocidad de Convergencia: Jacobi vs Gauss-Seidel');
    legend show;
    hold off;
end
```

.

.

Despues tenemos la funcion de generar sistema que es el encargado de devolver la matriz A y el vector rho dependiendo la cantidad de n que le pase el main.

```
function [A, rho] = generar_sistema(n)
    h = 1 / (n - 1);           % Paso
    A = zeros(n, n);           % Inicializar matriz A
    rho = zeros(n, 1);         % Inicializar vector rho

    % Definir la matriz tridiagonal A
    for i = 2:n-1
```

```
        A(i, i-1) = -1;        % Diagonal inferior
        A(i, i) = 2;           % Diagonal principal
        A(i, i+1) = -1;        % Diagonal superior
    end

    % Condiciones de borde
    A(1, 1) = 1;               % Primera fila
    A(n, n) = 1;               % Última fila

    % Definir el vector independiente rho
    for j = 2:n-1
        xj = h * (j - 1);
        rho(j) = h^2 * (-xj * (xj + 3) * exp(xj));
    end

    % Condiciones de borde para rho
    rho(1) = 0;
    rho(n) = 0;
end
```

Una vez que tenemos estos datos ya podemos hablar con las dos funciones encargadas de iterar, Gauss Seidel y Jacobi

Gauss Seidel:

```
function [phi, error_hist] = gauss_seidel(A, rho, tol, max_iter)
    n = length(rho);           % Tamaño del sistema
    phi = ones(n, 1);          % Vector inicial de soluciones
    error_hist = zeros(max_iter, 1); % Historial de errores

    for iter = 1:max_iter
        for j = 1:n
            resta = 0;

            for k = 1:j-1
                resta = resta + A(j, k) * phi(k);
            end

            for k = j+1:n
                resta = resta + A(j, k) * phi(k);
            end

            phi(j) = (rho(j) - resta) / A(j, j);
        end

        % Calcular el error
        error_hist(iter) = norm(A*phi - rho, inf);

        % Comprobar la convergencia
        if error_hist(iter) < tol
            error_hist = error_hist(1:iter);  % Truncar historial al número de iteraciones reales
            break;
        end
    end
end
```

Jacobi:

```
function [phi, error_hist] = jacobi(A, rho, tol, max_iter)
```

```
    n = length(rho);
    phi = ones(n, 1);        % Inicialización del vector de soluciones
    phi_new = phi;           % Inicialización para almacenar las nuevas iteraciones
    error_hist = zeros(max_iter, 1); % Historial de errores

    for k = 1:max_iter
        for i = 1:n
            resta = 0;
            for j = 1:n
                if j != i
                    resta = resta + A(i, j) * phi(j);
                end
            end
            phi_new(i) = (rho(i) - resta) / A(i, i);
        end

        % Calcular el error
        error_hist(k) = norm(phi_new - phi, inf);

        % Verificar convergencia
        if error_hist(k) < tol
            error_hist = error_hist(1:k);  % Truncar historial al número de iteraciones reales
            break;
        end

        phi = phi_new;  % Actualizar la solución
    end
end
```

Por Ultimo tenemos que analizar las "performance"de cada forma a travez de comparar que pasa con distintos tamaños y con distintos criterios de corte, para eso tenemos las funciones de comparar performance por tamaños y comparar performance por tolerancia.

Comparar performance por tamaños:

```
function comparar_performance_por_tamanos()
    tol = 0.00001;
    max_iter = 100000;
    tamanos = [11, 51, 101];  % Tamaños del sistema

    for i = 1:3
        n = tamanos(i);  % Usar paréntesis para acceder a los elementos del array
        fprintf('--- Sistema de tamaño n = %d ---\n', n);

        % Generar el sistema
        [A, rho] = generar_sistema(n);
        init_value = 1;  % Solución inicial con todos los elementos en 1

        % Medir tiempo de Jacobi
        tic;
        [phi_jacobi, iter_jacobi] = jacobi(A, rho, tol, max_iter);
        tiempo_jacobi = toc;
        fprintf('Jacobi tomó %f segundos y %d iteraciones.\n', tiempo_jacobi, length(iter_jacobi));

        % Medir tiempo de Gauss-Seidel
        tic;
        [phi_gauss, iter_gauss] = gauss_seidel(A, rho, tol, max_iter);
        tiempo_gauss = toc;
        fprintf('Gauss-Seidel tomó %f segundos y %d iteraciones.\n', tiempo_gauss, length(iter_gauss));
```

```
        % Comparar soluciones
        error_jacobi = norm(A*phi_jacobi - rho, inf);
        error_gauss = norm(A*phi_gauss - rho, inf);
        fprintf('Error Jacobi: %e | Error Gauss-Seidel: %e\n\n', error_jacobi, error_gauss);
    end
end
```

Comparar performance por tolerancia:

```
function comparar_performance_por_tolerancia()
    max_iter = 10000;
    tolerancias = [0.00001, 0.0001, 0.001];  % Diferentes tolerancias
    n = 101;

    for i = 1:length(tolerancias)
        tol = tolerancias(i);  % Usar paréntesis para acceder a los elementos del array
        fprintf('--- Sistema con tolerancia = %e ---\n', tol);  % Mostrar tolerancia en formato científ

        % Generar el sistema
        [A, rho] = generar_sistema(n);
        init_value = 1;  % Solución inicial con todos los elementos en 1

        % Medir tiempo de Jacobi
        tic;
        [phi_jacobi, iter_jacobi] = jacobi(A, rho, tol, max_iter);
        tiempo_jacobi = toc;
        fprintf('Jacobi tomó %f segundos y %d iteraciones.\n', tiempo_jacobi, length(iter_jacobi));

        % Medir tiempo de Gauss-Seidel
        tic;
        [phi_gauss, iter_gauss] = gauss_seidel(A, rho, tol, max_iter);
        tiempo_gauss = toc;
        fprintf('Gauss-Seidel tomó %f segundos y %d iteraciones.\n', tiempo_gauss, length(iter_gauss));

        % Comparar soluciones
        error_jacobi = norm(A*phi_jacobi - rho, inf);
        error_gauss = norm(A*phi_gauss - rho, inf);
        fprintf('Error Jacobi: %e | Error Gauss-Seidel: %e\n\n', error_jacobi, error_gauss);
    end
end
```