Comparación del rendimiento y convergencia de métodos de solución de ecuaciones lineales implementados en Julia y Python

CÓMPUTO DE ALTO RENDIMIENTO CON LENGUAJES DE ALTO NIVEL

Profesor: Dr. Oscar A. Esquivel Flores

Alumno: Miguel Ángel Veloz Lucas

Objetivo:

Utilizar lenguajes de alto rendimiento para evaluar eficiencia al implementar estrategias numéricas para resolver sistemas de ecuaciones lineales.

Generalidades

Rendimiento y convergencia

- Se utilizarán dos lenguajes de alto nivel para este proyecto: python y julia.

- Se utilizará material visto en clase para el funcionamiento de algunos métodos de solución de sistemas de ecuaciones lineales e indagar si la implementación impacta en el rendimiento como una necesidad más que una alternativa de programación.

- Este microproyecto nos acercará a considerar el cómputo de alto rendimiento como una necesidad más que una alternativa de programación.

1. Resumen

En la computación entendemos el rendimiento como la medida o cuantificación de la velocidad/resultado con que se realiza una tarea o proceso. En una computadora, su rendimiento no depende sólo del microprocesador como suele pensarse, sino de la suma de sus componentes, sus softwares y la configuración de estos. Las mediciones de rendimiento pueden estar orientadas al usuario (tiempos de respuesta) o hacia el sistema (utilización de la recursos).

Al igual que la seguridad de las aplicaciones, la temática de rendimiento debe ser contemplada desde el inicio del diseño de los sistemas de información considerando las directrices que permitan alinear las infraestructuras y necesidades específicas de los sistemas en desarrollo, con las directrices y recomendaciones de esta área.

Índice

1. Resumen
2. Introducción
3. Función para matrices de prueba Ax=b
4. Análisis de convergencia, implementación de métodos en julia
    a. Jacobi
    b. Gauss-Seidel
    c. Sobre-relajación sucesiva
    d. Gradiente conjugado
5. Análisis de rendimiento, implementación de métodos en julia y python
    a. Jacobi
    b. Gauss-Seidel
    c. Sobre-relajación sucesiva
    d. Gradiente conjugado
6. Conclusiones y Referencias

2. Introducción

Los métodos denominados iterativos también pueden aplicarse a la resolución de sistemas de ecuaciones lineales algebraicos. De manera similar a la iteración de punto fijo , los métodos comienzan con un valor inicial y mejoran la estimación en cada paso , convergiendo hacia el vector solución del sistema.

2. Función para matrices de prueba Ax=b

La implementación de esta función tiene como objetivo construir sistemas de ecuaciones ***Ax=b*** de manera que pueda ser escalada la matriz de prueba ingresando solo como argumento el tamaño ***n*** para obtener la matriz ***A*** de dimensiones ***n x n*** y el vector ***b*** de dimensiones ***n***.

$$A = \begin{bmatrix} 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & -1 & 3 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 \end{bmatrix}.$$

Imagen: Sauer T., “Análisis Numérico”, segunda edición, Pearson, pp. 114.

Implementación en lenguaje julia

Una matriz de coeficientes se llama **dispersa** si se sabe que muchas de las entradas de la matriz son iguales a cero.

```
function matriz_dispersa(n)
    e = ones(n)
    n2 =Int(n/2)
    diags = [-1,0,1]
    A = Matrix(spdiagm(-1 => -ones(n-1)
        ,0 => 3*ones(n),1 => -ones(n-1)))
    c = spdiagm(0 => ones(n)/2)
    ab = [x for x=1:n]
    ba = [(n+1)-x for x=1:n]
    c = Matrix(permute(c, ba, ab))

    A = A + c
    A[n2+1,n2] = -1
    A[n2,n2+1] = -1

    b = zeros(n,1)
    b[1] = 2.5
    b[n] = 2.5
    b[2:n-1] .= [1.5]
    b[n2:n2+1] .= [1]
    x0 = zeros(n)
    return A,b,x0
end
```

Implementación en lenguaje python

Una matriz **llena** es lo opuesto , donde algunas entradas pueden ser iguales a cero.

```
def matriz_dispersa(n):
    e = np.ones(n)

    n2 = int(n/2)
    diags = [-1,0,1]
    A = scipy.sparse.spdiags(
        [-1*e, 3*e ,-1*e],
        diags ,n,n).toarray()
    c = scipy.sparse.spdiags(
        [e/2],0,n,n).toarray()
    c = np.fliplr(c)

    A = A + c
    A[n2-1][n2] = -1
    A[n2,n2-1]=-1
    b = np.zeros((n,1))
    b[0] = 2.5
    b[n-1] = 2.5
    b[1:n-1] = 1.5
    b[n2-1:n2+1] = 1

    x0 = np.zeros((n,1))
    return A,b,x0
```

Ejemplo de operación

Matriz dispersa: ***A*** de tamaño ***n=14***

```
25 A,b,x0 = matriz_dispersa(14)
26 print(A)
```

```
[[ 3.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.5]
 [-1.   3.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.5  0. ]
 [ 0.  -1.   3.  -1.   0.   0.   0.   0.   0.   0.   0.5  0.   0. ]
 [ 0.   0.  -1.   3.  -1.   0.   0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.   0.   0.  -1.   3.  -1.   0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.   0.   0.   0.  -1.   3.  -1.   0.   0.5  0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.  -1.   3.  -1.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.  -1.   3.  -1.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.5  0.  -1.   3.  -1.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.5  0.   0.   0.  -1.   3.  -1.   0.   0. ]
 [ 0.   0.   0.   0.5  0.   0.   0.   0.   0.  -1.   3.  -1.   0. ]
 [ 0.   0.   0.5  0.   0.   0.   0.   0.   0.   0.  -1.   3.  -1. ]
 [ 0.   0.5  0.   0.   0.   0.   0.   0.   0.   0.   0.  -1.   3. ]]
```

4. Análisis de convergencia

IMPLEMENTACIÓN DE MÉTODOS CON LENGUAJE JULIA

a. Método de Jacobi

Este método es una forma de iteración de punto fijo para un sistema de ecuaciones. En la iteración de punto fijo el primer paso consiste en reescribir las ecuaciones, despejando las incógnitas. El primer paso del método de Jacobi es hacer esto de la forma estándar que se indica a continuación: a) resolver la i−ésima ecuación de la i−ésima incógnita, b)Iterar como en la iteración de punto fijo a partir de una estimación inicial.

Implementación en lenguaje julia

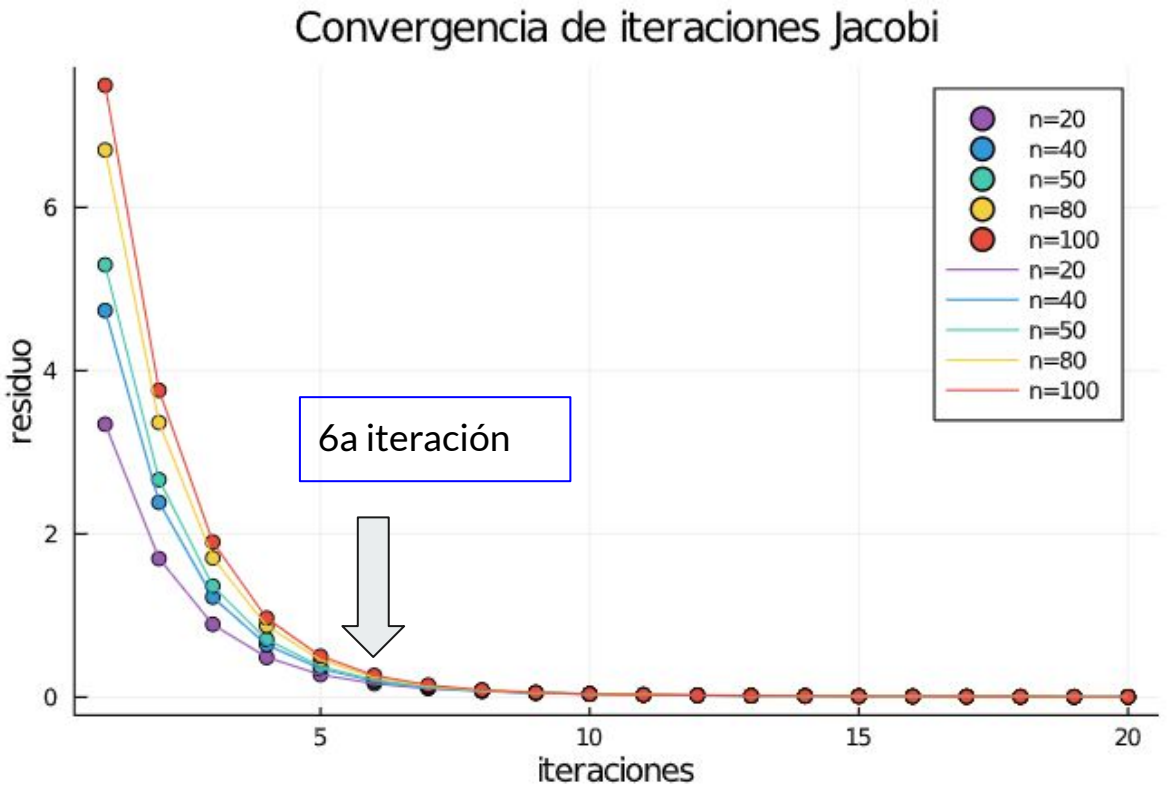
Los resultados en esta función van a estar dados por la matriz normres_jacobi[ ].

```
function jacobi_jl(A,b,x0,kn)
    D = Diagonal(A)
    U = triu(A, 1)
    L = tril(A, -1)
    normres_jacobi = [];
    x = x0
    m = 0
    for k = 1:kn
        x = inv(D)*(b-(L + U)*x);
        normres_jacobi = [normres_jacobi;
            norm(b-A*x)];
    end
    return normres_jacobi
end
```

Convergencia Jacobi

Prueba realizada con 20 iteraciones para matrices dispersas ***A*** de tamaño ***n={20,40,50,80,100}.***

Con este método parece acercarse a la solución partir de la octava o novena iteración.



4. Análisis de convergencia

IMPLEMENTACIÓN DE MÉTODOS CON LENGUAJE JULIA

b. Método de Gauss-Seidel

En estrecha relación con el método de Jacobi existe una iteración llamada método de Gauss-Seidel. La única diferencia entre este método y el de Jacobi es que en el de Gauss-Seidel los valores más recientemente utilizados de las incógnitas se utilizan en cada paso, incluso si la actualización se produce en el paso actual.

Implementación en lenguaje julia

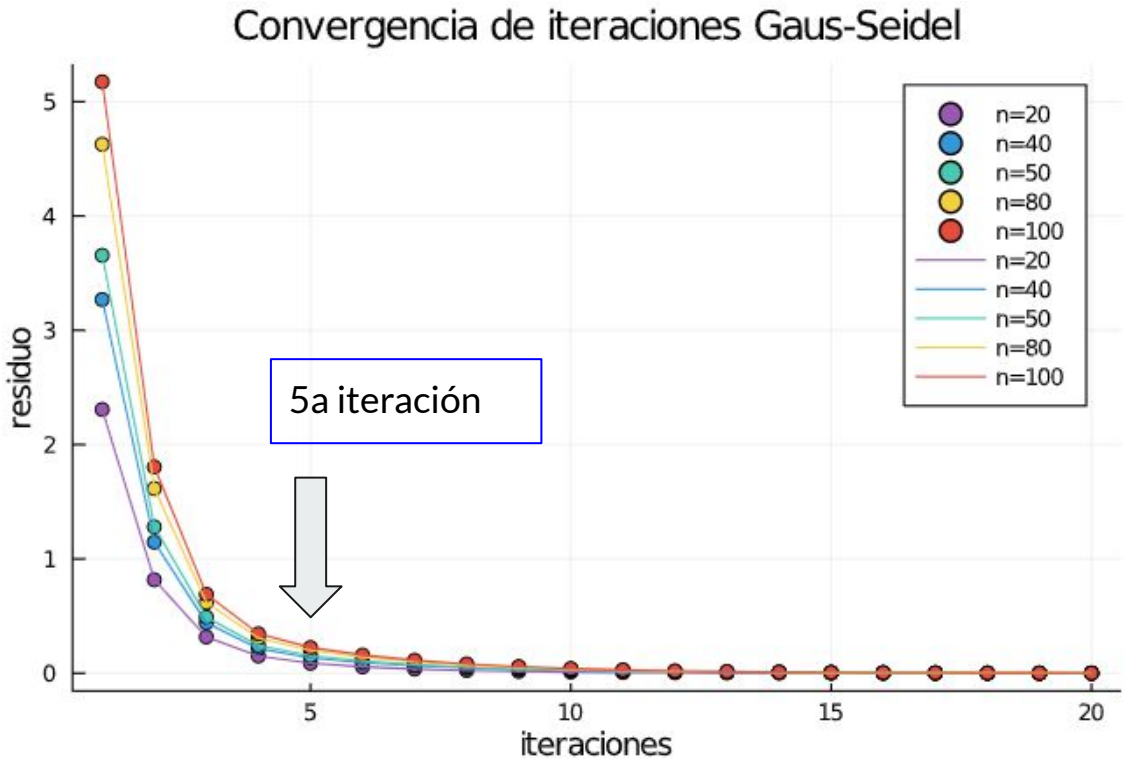
La aproximación del este método es más exacta que la de Jacobi en el mismo número de pasos. Con frecuencia el método de Gauss-Seidel converge más rápido que el método de Jacobi cuando éste converge.

```
function gauss_seidel_jl(A,b,x0,kn)
    L = tril(A)
    U = triu(A,1)
    x0 = zeros(size(b))
    normres_gauss = [];
    x = x0
    l=0
    for k = 1:kn
    x = inv(L)*(b-U*x);100
    normres_gauss = [normres_gauss;norm(b-A*x)]

        if norm(b-A*x) <= 1e-10
            l=k
            break
        end
    end
    return normres_gauss
end
```

Convergencia Gauss-Seidel

Como se mencionó antes, este método se acerca mucho a la solución del sistema, reduciendo rápidamente el error, alrededor de quinta iteración, mostrando mejor eficiencia que el método de Jacobi.



4. Análisis de convergencia

IMPLEMENTACIÓN DE MÉTODOS CON LENGUAJE JULIA

c. Método de Sobre-relajación sucesiva

Este método toma la dirección de Gauss-Seidel hacia la solución y lo “rebasa” para tratar de acelerar la convergencia. Sea w un número real y defina cada componente de la nueva estimación x k + 1 como la media ponderada de ω veces la fórmula de Gauss-Seidel y 1 − ω veces la estimación actual de x k . El número ω se conoce como el parámetro de relajación, y ω > 1 se conoce como sobre-relajación.

Implementación en lenguaje julia

Al igual que el método de Jacobi y Gauss-Seidel, en este método es posible una derivación alternativa de la SRS si se trata al sistema como un problema de punto fijo.

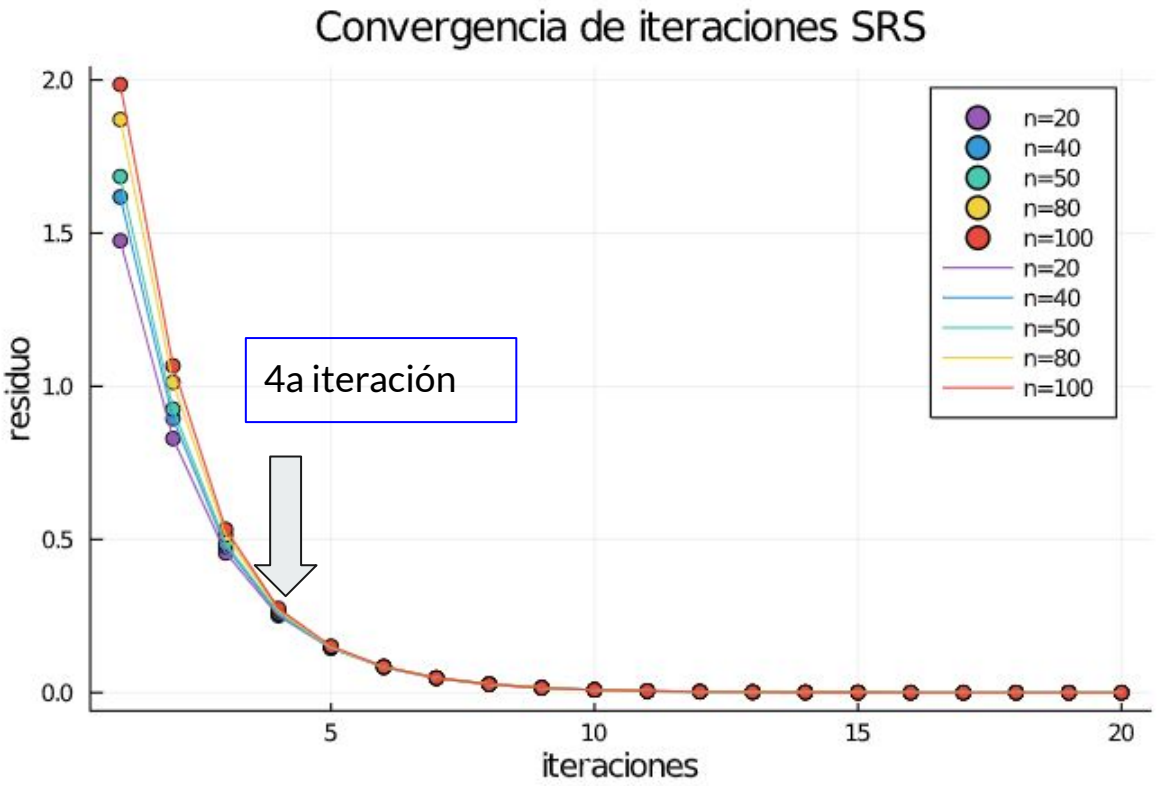
```
function srs_jl(A,b,x0,kn,w)
    U = triu(A,1)
    L = tril(A,-1)
    D = Diagonal(A)
    normres_srs = []
    x = x0
    for k = 1:kn
        x = inv(w * L + D) *
        ((1 - w) * D * x - (w * U * x)) +
        (w * inv(D + L * w) * b)
        normres_srs = [normres_srs;norm(b-A*x)]

    end
    return normres_srs

end
```

Convergencia SRS

En este método la reducción del error es notable desde el inicio, en comparación con los métodos anteriores, y se logra acercar al resultado desde la cuarta iteración, mostrando una mayor eficiencia en la convergencia.



4. Análisis de convergencia

IMPLEMENTACIÓN DE MÉTODOS CON LENGUAJE JULIA

d. Método de Gradiente conjugado

La iteración del gradiente conjugado actualiza tres vectores diferentes en cada paso. El vector xk es la solución aproximada en el paso k. El vector rk representa el residuo de la solución aproximada xk. Por último, el vector dk representa la nueva dirección de búsqueda que se utiliza para actualizar la aproximación xk con la versión mejorada xk+1. El método tiene éxito porque cada residuo está dispuesto para ser ortogonal a todos los residuos previos. Si esto puede hacerse, el método se ejecuta de las direcciones ortogonales en las cuales buscar, y debe llegarse a un residuo cero y a una solución correcta cuando mucho en n pasos.

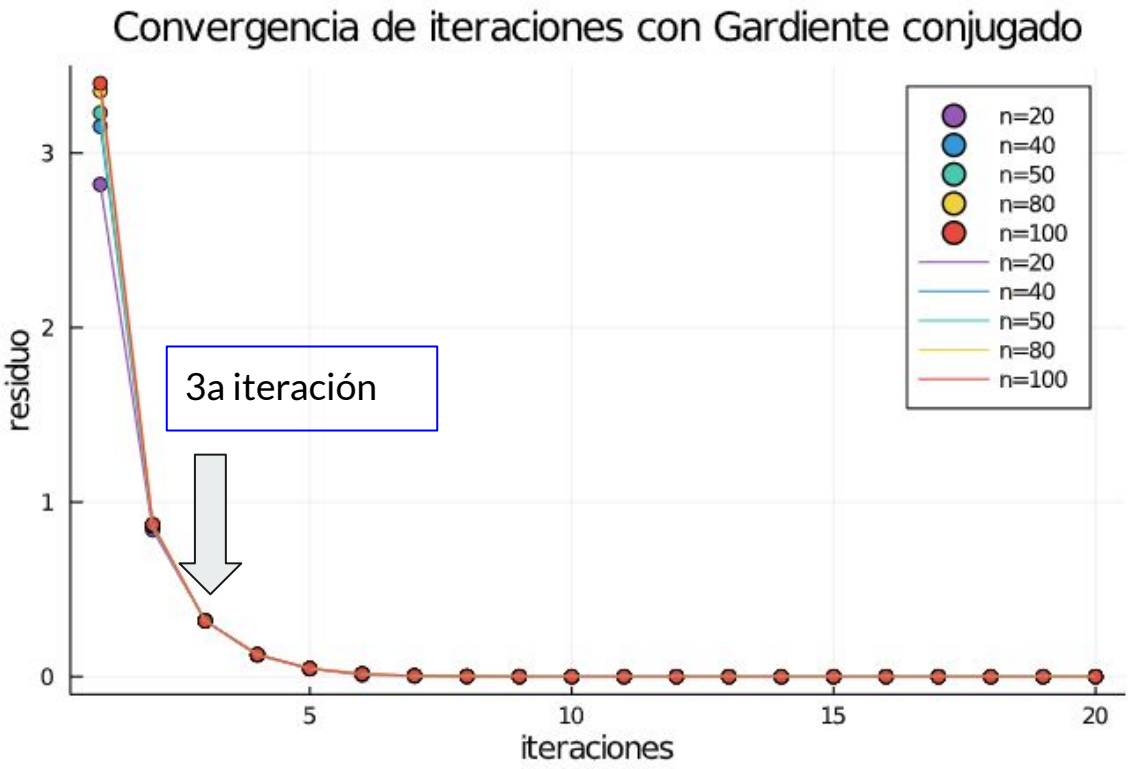
Implementación en lenguaje julia

La clave para lograr la ortogonalidad entre los residuos resulta ser la elección de las direcciones de búsqueda dk por pares conjugados. El concepto de conjugación generaliza la ortogonalidad y da nombre al algoritmo.

```
function gcd_jl(A,b,x0,kn)
     normres_grc = []
    x = x0
    r = b - A * x
    d = copy(r)
    alfa=10
    for k = 1:kn
        alfa = (r'*r)*(d'*A*d)^(-1)
        x = x + (alfa[1]) * d
        rk = r - alfa[1] * A * d
        beta = (transpose(rk) * rk) * (transpose(r)*r)^(-1)
        dk = rk + beta[1] * d
        r = rk
        d = dk
        normres_grc =[normres_grc;norm(b - A * x)]
    end
    return normres_grc
end
```

Convergencia GDC

Con este método la aproximación a la solución sucede desde la tercera iteración, mostrando una reducción drástica del residuo desde el inicio de las repeticiones, marcando una eficiencia mayor a los métodos anteriores.



5. Análisis de rendimiento

IMPLEMENTACIÓN DE MÉTODOS CON LENGUAJE JULIA PYTHON

Para la realización de este análisis se implementaron llamadas a funciones de python utilizando la biblioteca **PyCall** para extraer el detalle de los tiempo de ejecución con **BenchmarkTools**. En todos los casos se utilizaron un número de 20 iteraciones para cada uno de los módelos iterativo, con sistemas de tamaño $n \in$ **(20,40,50,80,100).**

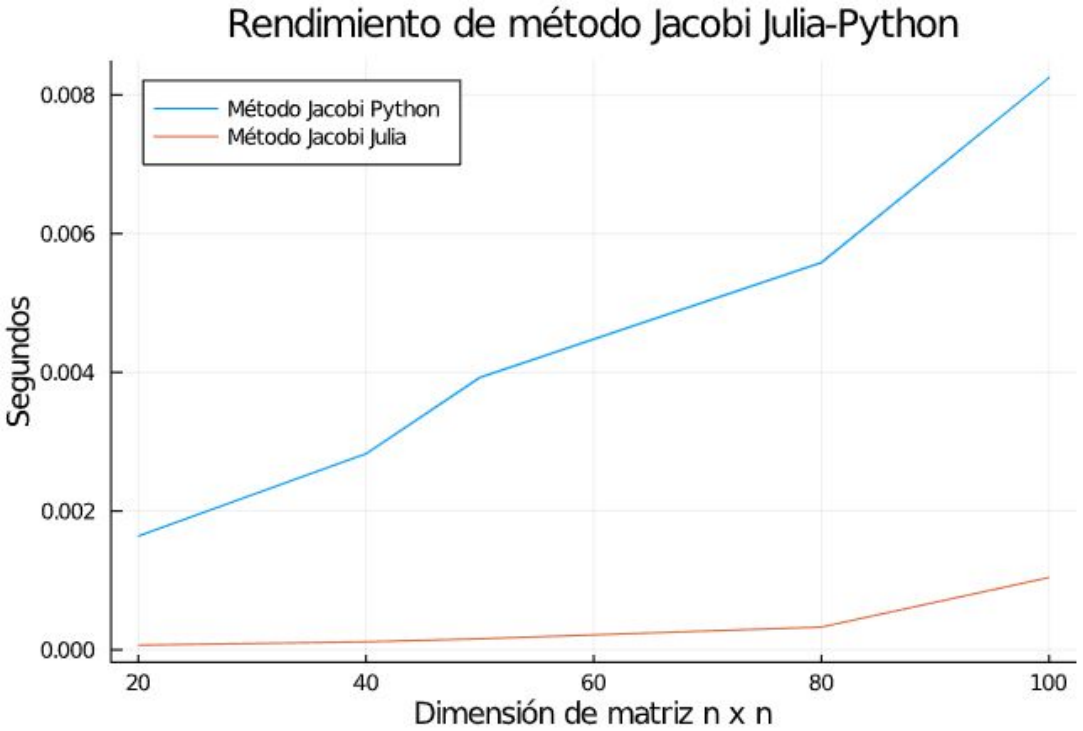
a. Análisis de rendimiento Jacobi

```
yJacobiP = []
yJacobiJ = []
for i = 1:ln
    n = N[i]
    A,b,x0 = matriz_dispersa(n)
    jP = @benchmark py"jacobi_py"(A,b,x0,k) seconds=1
    jJ = @benchmark jacobi_jl(A,b,x0,k) seconds=1
    push!(yJacobiJ, jJ)
    push!(yJacobiP, jP)
end
```

En este bloque de código de julia se observar la llamada a la función **jacobi_py()** y **jacobi_jl()** para obtener las métricas de tiempo de ejecución para cada uno de los distintos tamaños de **n**.

Método Jacobi

n	*Tiempo mínimo*	
	Python	*Julia*
20	1.636 ms	67.021 µs
40	425 ms	115.182 µs
50	527 ms	158.634 µs
80	882 ms	327.954 µs
100	1250 ms	1.043 ms



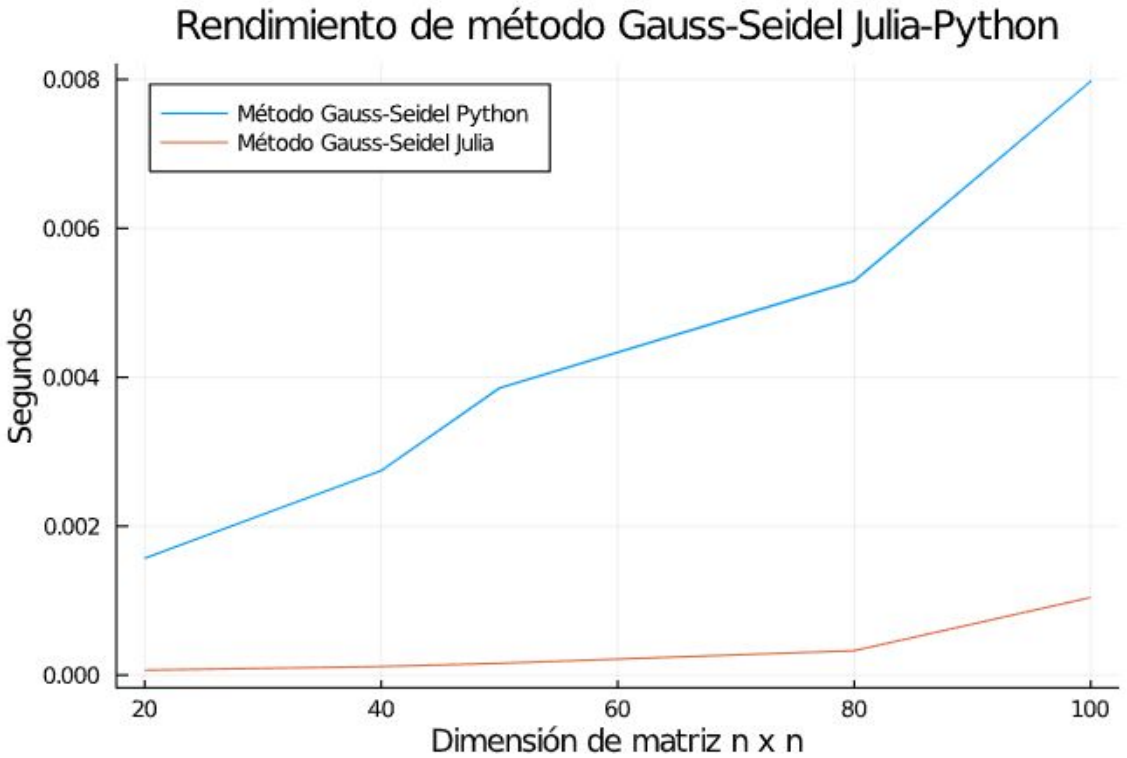
b. Análisis de rendimiento Gauss-Seidel

```
yGaussSP = []
yGaussSJ = []
for i = 1:ln
    n = N[i]
    A,b,x0 = matriz_dispersa(n)
    gP = @benchmark py"gauss_seidel_py"(A,b,x0,k) seconds=1
    gJ = @benchmark gauss_seidel_jl(A,b,x0,k) seconds=1
    push!(yGaussSP, gP)
    push!(yGaussSJ, jJ)
end
```

Código de julia donde se observar la llamada a la función **gauss_seidel_jl()** y **gauss_seidel_py()** para obtener las métricas de tiempo de ejecución para cada uno de los distintos tamaños de **n**.

Método Gauss-Seidel

n	*Tiempo mínimo*	
	Python	*Julia*
20	567 ms	67.021 µs
40	2.745 ms	67.021 µs
50	3.854 ms	158.634 µs
80	5.295 ms	327.954 µs
100	7.976 ms	1.043 ms



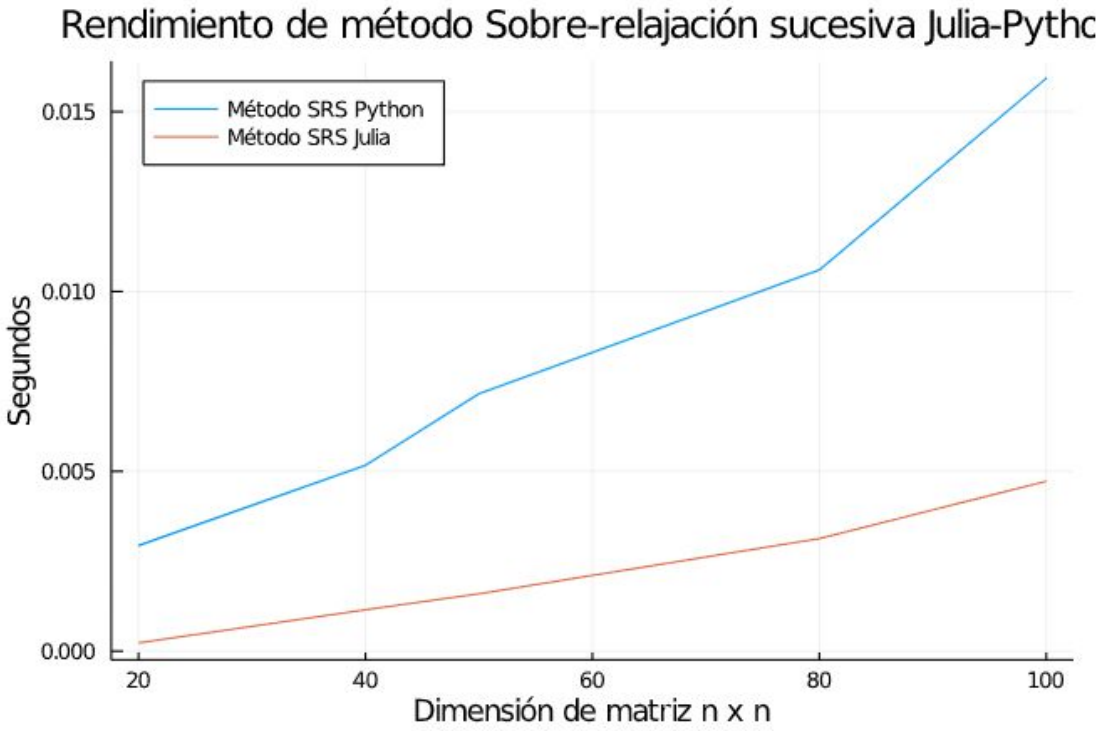
c. Análisis de rendimiento Sobre-relajación sucesiva

```
ySRSP = []
ySRSJ = []
for i = 1:ln
    n = N[i]
    A,b,x0 = matriz_dispersa(n)
    sP = @benchmark py"srs_py"(A,b,x0,k,w) seconds=1
    sJ = @benchmark srs_jl(A,b,x0,k,w) seconds=1
    push!(ySRSP, sP)
    push!(ySRSJ, sJ)
end
```

Bloque de código de julia donde se observar la llamada a la función **srs_jl()** y **srs_py()** para obtener las métricas de tiempo de ejecución para cada uno de los distintos tamaños de **n**.

Método SRS

n	Tiempo mínimo	
	Python	Julia
20	2.936 ms	228.976 µs
40	5.170 ms	1.150 ms
50	7.159 ms	1.594 ms
80	10.597 ms	3.129 ms
100	15.932 ms	4.724 ms



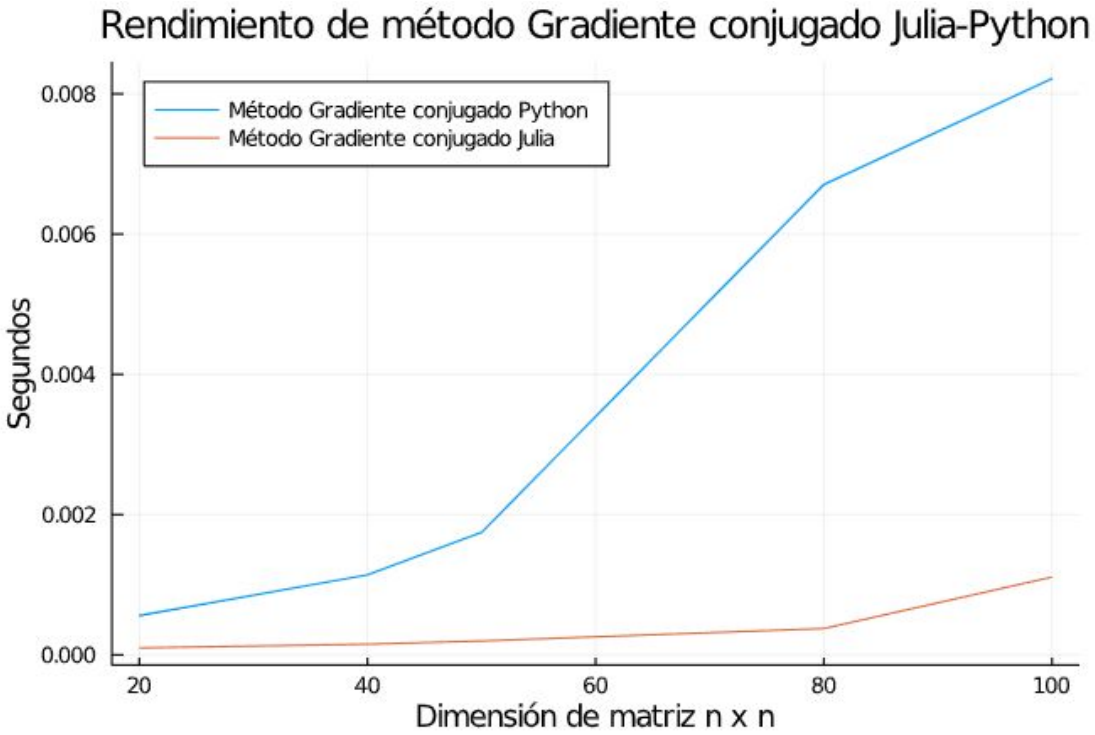
d. Análisis de rendimiento Gradiente conjugado

```
yGCDP = []
yGCDJ = []
for i = 1:ln
    n = N[i]
    A,b,x0 = matriz_dispersa(n)
    gcP = @benchmark py"gcd_py"(A,b,x0,k) seconds=1
    gcJ = @benchmark gcd_jl(A,b,x0,k) seconds=1
    push!(yGCDP, gcP)
    push!(yGCDJ, gcJ)
end
```

Código de julia donde se observar la llamada a la función **gcd_py()** y **gcd_py()** para obtener las métricas de tiempo de ejecución para cada uno de los distintos tamaños de **n**.

Método Gradiente conjugado

n	*Tiempo mínimo*	
	Python	*Julia*
20	558.385 µs	98.688 µs
40	1.141 ms	151.984 µs
50	1.747 ms	197.498 µs
80	6.709 ms	374.996 µs
100	8.215 ms	1.109 ms



Tiempos obtenidos

n	Jacobi		Gauss-Seidel		SRS		GDC	
	Python	Julia	Python	Julia	Python	Julia	Python	Julia
20	1.636 ms	67.021 µs	567 ms	67.021 µs	2.936 ms	228.976 µs	558.385 µs	98.688 µs
40	425 ms	115.182 µs	2.745 ms	67.021 µs	5.170 ms	1.150 ms	1.141 ms	151.984 µs
50	527 ms	158.634 µs	3.854 ms	158.634 µs	7.159 ms	1.594 ms	1.747 ms	197.498 µs
80	882 ms	327.954 µs	5.295 ms	327.954 µs	10.597 ms	3.129 ms	6.709 ms	374.996 µs
100	1250 ms	1.043 ms	7.976 ms	1.043 ms	15.932 ms	4.724 ms	8.215 ms	1.109 ms

6. Conclusiones

De acuerdo a los resultados obtenidos con los ejercicios de este proyecto, es muy notable la velocidad de ejecución que ofrece julia en operaciones iterativas. Si bien las funciones ejecutadas de python realizan la misma labor a una velocidad aceptable, se vuelve relevante la eficiencia y el rendimiento en la ejecución de operaciones de grandes bloques de datos, desde matrices numéricas hasta dataset para clasificación de datos o aprendizaje automatizado. La implementación práctica de estos ejercicios sobre métodos iterativos, contribuyen a un entendimiento más sólido en cuanto a velocidad y rendimiento computacional se refiere.

Referencias

- Sauer T., “Análisis Numérico”, segunda edición, Pearson, 2013
- Nolasco Valenzuela JS. Python : Aplicaciones Prácticas. Primera edición. Ediciones de la U; 2018
- https://opensourc.es/blog/benchmarking-and-profiling-julia-code/
- https://docs.julialang.org
- https://juliacomputing.com/blog/2020/06/fast-csv/
- https://het.as.utexas.edu/HET/Software/Scipy/generated/scipy.sparse.spdiags.html
- https://datapeaker.com/big-data/k-vecino-mas-cercano-algoritmo-knn/