Análisis computacional

```
begin
    using LinearAlgebra
    using BenchmarkTools
    using Plots
end
```

Implementación

Descomposición QR

Antiguo código

Partimos del código de Givens trabajado en la última tarea para mejorarlo:

```
no_shift_givens_qr (generic function with 1 method)
```

```julia
function no_shift_givens_qr(A)
    A = copy(Matrix(A))  # Trabajamos con una copia densa para modificarla en
sitio
    m, n = size(A)
    Q = Matrix(1.0I, m, m)  # Acumulador ortogonal

    for j in 1:n
        for i in m:-1:(j + 1)
            a = A[i-1, j]
            b = A[i, j]
            c, s = givens(a, b)

            # Construir Givens implícitamente
            # y aplicar a filas i-1 e i de A (desde columna j en adelante)
            for k in j:n
                temp1 = c * A[i-1, k] + s * A[i, k]
                temp2 = -s * A[i-1, k] + c * A[i, k]
                A[i-1, k] = temp1
                A[i, k]   = temp2
            end

            # Acumular Givens en Q
            for k in 1:m
                temp1 = c * Q[k, i-1] + s * Q[k, i]
                temp2 = -s * Q[k, i-1] + c * Q[k, i]
                Q[k, i-1] = temp1
                Q[k, i]   = temp2
            end
        end
    end

    R = triu(A)
    return Q, R
end
```

Funciones auxiliares

Lo modularizamos para mejorar su claridad. Tenemos cuidado de tratar con `views` para no tener copias temporales.

apply_shift!

```
apply_shift!(A, μ)
```

Resta el desplazamiento μ de la diagonal de A, in-place.

```
"""
    apply_shift!(A, μ)

Resta el desplazamiento μ de la diagonal de A, in-place.
"""
function apply_shift!(A, μ)
    for i in 1:min(size(A)...)
        A[i, i] -= μ
    end
end
```

compute_givens

```
compute_givens(a, b)
```

Devuelve los coseno y seno (c, s) para anular b con una rotación de Givens.

```
"""
    compute_givens(a, b)

Devuelve los coseno y seno (c, s) para anular b con una rotación de Givens.
"""
function compute_givens(a, b)
    if b == 0
        return 1.0, 0.0
    else
        r = hypot(a, b)
        c = a / r
        s = b / r
        return c, s
    end
end
```

rotate_rows!

```
rotate_rows!(A, i, c, s, j_range)
```

Aplica la rotación de Givens a las filas `i-1` e `i` de la matriz A sobre las columnas en `j_range`.

```
"""
    rotate_rows!(A, i, c, s, j_range)

Aplica la rotación de Givens a las filas `i-1` e `i` de la matriz A sobre las columnas en `j_range`.
"""
function rotate_rows!(A, i, c, s, j_range::UnitRange)
    @views row1 = A[i-1, j_range]
    @views row2 = A[i, j_range]

    for k in eachindex(row1)
        temp1 =  c * row1[k] + s * row2[k]
        temp2 = -s * row1[k] + c * row2[k]
        row1[k] = temp1
        row2[k] = temp2
    end
end

```

accumulate_q!

```
accumulate_q!(Q, i, c, s)
```

Aplica la rotación de Givens a las columnas `i-1` e `i` de la matriz ortogonal Q.

```
"""
    accumulate_q!(Q, i, c, s)

Aplica la rotación de Givens a las columnas `i-1` e `i` de la matriz ortogonal Q.
"""
function accumulate_q!(Q, i, c, s)
    @views col1 = Q[:, i-1]
    @views col2 = Q[:, i]

    for k in eachindex(col1)
        temp1 =  c * col1[k] + s * col2[k]
        temp2 = -s * col1[k] + c * col2[k]
        col1[k] = temp1
        col2[k] = temp2
    end
end
```

```
subdiagonal_norm (generic function with 1 method)
```

```
function subdiagonal_norm(A::AbstractMatrix)
    n = size(A, 1)
    return norm([A[i+1, i] for i in 1:n-1])
end

```

Función principal

`givens_qr`

```
givens_qr(A)
```

Devuelve las matrices Q y R tales que A ≈ QR usando rotaciones de Givens. Puede aplicar un desplazamiento o una reducción previa a forma de Hessenberg.

```
"""
    givens_qr(A)

Devuelve las matrices Q y R tales que A ≈ QR usando rotaciones de Givens.
Puede aplicar un desplazamiento o una reducción previa a forma de Hessenberg.
"""
function givens_qr(A)
    A = copy(A)
    m, n = size(A)
    Q_matrix = Matrix{eltype(A)}(I, m, m)

    for j in 1:n
        for i in m:-1:(j+1)
            a, b = A[i-1, j], A[i, j]
            c, s = compute_givens(a, b)
            rotate_rows!(A, i, c, s, j:n)
            accumulate_q!(Q_matrix, i, c, s)
        end
    end

    return Q_matrix, triu(A)
end
```

Algoritmos QR prácticos implementados

Se han definido dos funciones principales para ejecutar variantes del algoritmo QR en la práctica:

`qr_algorithm`: Esta función permite aplicar el algoritmo QR en tres modalidades, seleccionadas mediante argumentos opcionales:

- Versión básica sin preprocesamiento,
- Con reducción previa a forma de Hessenberg,
- O con shift estático aplicado a la matriz Hessenberg.

`qr_algorithm_with_dynamic_shift`: Esta función aplica el algoritmo QR con shift dinámico, actualizado en cada iteración según una estrategia proporcionada (por ejemplo, Rayleigh o Wilkinson). La matriz se reduce previamente a forma de Hessenberg.

```
qr_algorithm (generic function with 1 method)
```

```julia
function qr_algorithm(A; pre_hessenberg=false, shift=0.0, tol=1e-10, maxiter=1000, verbose=false)
    n = size(A, 1)
    iteration = 0
    Id = Matrix{eltype(A)}(I, size(A)...)

    # Paso 0: Reducción a forma de Hessenberg si se solicita
    Ak = pre_hessenberg ? Matrix(hessenberg(A).H) : copy(A)

    # Aplicar shift inicial si corresponde
    if pre_hessenberg && shift != 0.0
        apply_shift!(Ak, shift)
    end

    # Iteraciones QR
    for k in 1:maxiter
        Q, R = givens_qr(Ak)
        Ak = R * Q

        subdiag = [Ak[i+1, i] for i in 1:n-1]
        verbose && println("Iteración $k, ||subdiag|| = ", norm(subdiag))

        if norm(subdiag) < tol
            iteration = k
            break
        end
    end

    # Verificación si no se alcanzó la convergencia
    max_iter_reached = (iteration == 0)

    # Deshacer shift si fue aplicado
    if pre_hessenberg && shift != 0.0
        apply_shift!(Ak, -shift)
    end

    return (Ak=Ak, iteration=max_iter_reached ? maxiter : iteration, max_iter_reached)
end
```

```
qr_algorithm_with_dynamic_shift (generic function with 1 method)
```

```
function qr_algorithm_with_dynamic_shift(A; shift_method, tol=1e-5, maxiter=1000, verbose=false)
    n = size(A, 1)
    iteration = 0

    Id = Matrix{eltype(A)}(I, size(A)...)

    # Paso 0: reducción a forma de Hessenberg (requerido)
    Ak = Matrix(hessenberg(A).H)

    for k in 1:maxiter
        u = shift_method(Ak)

        # Paso QR con shift: A - uI = QR ⇒ A⁺ = RQ + μI
        Q, R = givens_qr(Ak .- u * Id)
        Ak = R * Q .+ u * Id


        # Criterio de parada
        subdiag_norm = subdiagonal_norm(Ak)
        verbose && println("Iteración $k, ||subdiag|| = ", subdiag_norm)

        if subdiag_norm < tol
            iteration = k
            break
        end
    end

    max_iter_reached = (iteration == 0)
    return (Ak=Ak, iteration=max_iter_reached ? maxiter : iteration, max_iter_reached)
end

```

```
h_nn (generic function with 1 method)
```

```
# Funciones auxiliares de shift
h_nn(A) = A[end, end]
```

```
spectral_mean (generic function with 1 method)
```

```
spectral_mean(A) = tr(A) / size(A, 1)

# Función de prueba: matriz simétrica
```

Ejemplo

En este ejemplo, el único de los algoritmos que converge es el algoritmo de Rayleigh: `qr_algorithm_with_dynamic_shift(A, shift_method=h_nn)`. Podemos ver que los valores en su diagonal se acercan a sus autovalores.

```
A =
10×10 Matrix{Float64}:
 -4.84181      0.692781  -0.19327   -1.49879   …   1.46192    -1.26878   -1.31083
  0.692781    -0.398652  -1.52306   -1.82554       1.29843    -1.14515   -0.431599
 -0.19327     -1.52306   -0.1536     0.256472      0.208943   -0.224576  -1.05785
 -1.49879     -1.82554    0.256472  -4.04238      -0.560071   -1.46534   -1.49373
 -1.42501      0.657881  -1.25169    0.788841      0.0813731   0.463034  -2.45401
 -1.40491     -0.242346   1.9316    -2.37248   …   1.13515     1.69582   -0.285141
  0.00212786   0.802169  -1.30508   -0.915327     -0.102908    0.177574  -1.99628
  1.46192      1.29843    0.208943  -0.560071      0.791618   -0.115758  -1.51098
 -1.26878     -1.14515   -0.224576  -1.46534      -0.115758    1.44373    1.86195
 -1.31083     -0.431599  -1.05785   -1.49373      -1.51098     1.86195    0.538283
```

```
1 A = generate_symmetric(10)
```

```
1 display(eigvals(A))
```

```
10-element Vector{Float64}:
 -7.82158346340103
 -4.748195103574608
 -4.218713849853048
 -2.673364958781813
 -1.1065551583729984
  0.6188078157474391
  2.365038688899505
  2.733450693954206
  4.430808274583989
  5.602247294179672
```

```
(Ak = 10×10 Matrix{Float64}:                                                    , i
      -7.82158       -1.92355e-16   -1.53001e-15   …   4.00715e-17    5.26245e-17
```

```
1 qr_algorithm(A)
```

```
(Ak = 10×10 Matrix{Float64}:                                                    , iter
      -7.82158        1.90788e-15   -2.84462e-16   …  -2.63447e-16    5.61663e-19
```

```
1 qr_algorithm(A, pre_hessenberg=true)
```

```
(Ak = 10×10 Matrix{Float64}:                                                    , itera
      -7.82158       -8.13052e-16    2.6517e-16    …   7.57293e-16   -3.25114e-16
```

```
1 qr_algorithm_with_dynamic_shift(A, shift_method=h_nn)
```

```
(Ak = 10×10 Matrix{Float64}:
      -7.82158       -1.28445e-9     3.82529e-16    1.16706e-15   …  -1.30448e-16   -1.47
```

```
1 qr_algorithm_with_dynamic_shift(A, shift_method=spectral_mean)
```

Experimentos

Ahora, realizamos experimentos con la función `compare_qr_algorithms`. Esta función permite establecer:

- el número máximo de iteraciones,
- el método de generación y el tamaño de las matrices y
- el número de muestras por cada tamaño.

Trabajaremos con 5 variaciones del Algoritmo QR:

- básico: no_shift
- con reducción previa a hessenberg sin desplazamiento: hessenberg
- con desplazamiento estático: shift_static
- con desplazamiento de Rayleigh: shift_hnn
- con desplazamiento de Wilikinson: shift_mean

Luego graficamos los diferentes resultados con `plot_experiment`.

```
compare_qr_algorithms (generic function with 1 method)
```

```
function compare_qr_algorithms(ns; samples=3, maxiter=1000,
matrix_generator=generate_symmetric)
    n_sizes = length(ns)

    # Initialize result storage: one matrix per method
    results = (
        sizes = ns,
        original_matrix = Matrix{Any}(undef, n_sizes, samples),
        no_shift = Matrix{NamedTuple}(undef, n_sizes, samples),
        hessenberg = Matrix{NamedTuple}(undef, n_sizes, samples),
        shift_static = Matrix{NamedTuple}(undef, n_sizes, samples),
        shift_hnn = Matrix{NamedTuple}(undef, n_sizes, samples),
        shift_mean = Matrix{NamedTuple}(undef, n_sizes, samples),
    )

    for (i, n) in enumerate(ns)
        for j in 1:samples
            A = matrix_generator(n)

            results.original_matrix[i,j] = A

            results.no_shift[i, j] = qr_algorithm(A, verbose=false,
maxiter=maxiter)

            results.hessenberg[i, j] = qr_algorithm(
                A, verbose=false, pre_hessenberg=true, maxiter=maxiter
            )

            results.shift_static[i, j] = qr_algorithm(
                A, verbose=false, pre_hessenberg=true, shift=h_nn(A),
maxiter=maxiter
            )

            results.shift_hnn[i, j] = qr_algorithm_with_dynamic_shift(
                A, shift_method=h_nn, verbose=false, maxiter=maxiter
            )

            results.shift_mean[i, j] = qr_algorithm_with_dynamic_shift(
                A, shift_method=spectral_mean, verbose=false, maxiter=maxiter
            )
        end
    end

    return results
end
```

```
plot_experiment (generic function with 1 method)
```

```
function plot_experiment(experiment, func; ylabel="", title="", xlabel="Tamaño de la matriz", markers=true, y_max=nothing)
    plot(title=title, xlabel=xlabel, ylabel=ylabel)

    if y_max !== nothing
        plot!(ylim = (0, y_max))
    end

    for (name, data) in pairs(experiment)
        (name == :sizes || name == :original_matrix) && continue

        y = func(data, experiment.original_matrix)
        plot!(
            experiment.sizes,
            y,
            label = string(name),
            marker =:+
        )
    end
    plot!()
end

```

Realizamos dos experimentos que ejecutan los diferentes algoritmos en 5 matrices de cada tamaño. Uno en matrices simétricas y otro en matrices asimétricas con autovalores reales.

A partir de los resultados del experimento. Para cada tamaño vamos a graficar:

- el número de iteraciones promedio,
- la norma de la subdiagonal final,
- el error espectral.

```
generate_symmetric (generic function with 1 method)
```

```
function generate_symmetric(n)
    A = randn(n, n)
    return A + A'
end

# Comparador de algoritmos
```

```
generate_nonsymmetric_real_spectrum (generic function with 1 method)
```

```
function generate_nonsymmetric_real_spectrum(n)
    Λ = Diagonal(randn(n))  # reales
    Q, _ = qr(randn(n, n))
    return Q * Λ * inv(Q)
end

```

```
ns = 10:10:100
```

```
ns = 10:10:100  # Tamaños de matrices
```

```
symmetric_experiment =
  (sizes = 10:10:100, original_matrix = 10×5 Matrix{Any}:
                                        [1.56032 -0.344284 … 1.71254 -0.928766; -0.3442
```

```
1 symmetric_experiment = compare_qr_algorithms(ns, samples=5, maxiter=10000)
```

```
nonsymmetric_experiment =
  (sizes = 10:10:100, original_matrix = 10×5 Matrix{Any}:
                                        [-0.138484 -0.0710904 … -0.0422518 0.242569; -0
```

```
1 nonsymmetric_experiment = compare_qr_algorithms(ns, samples=5, maxiter=10000,
  matrix_generator = generate_nonsymmetric_real_spectrum)
```

Número de iteraciones

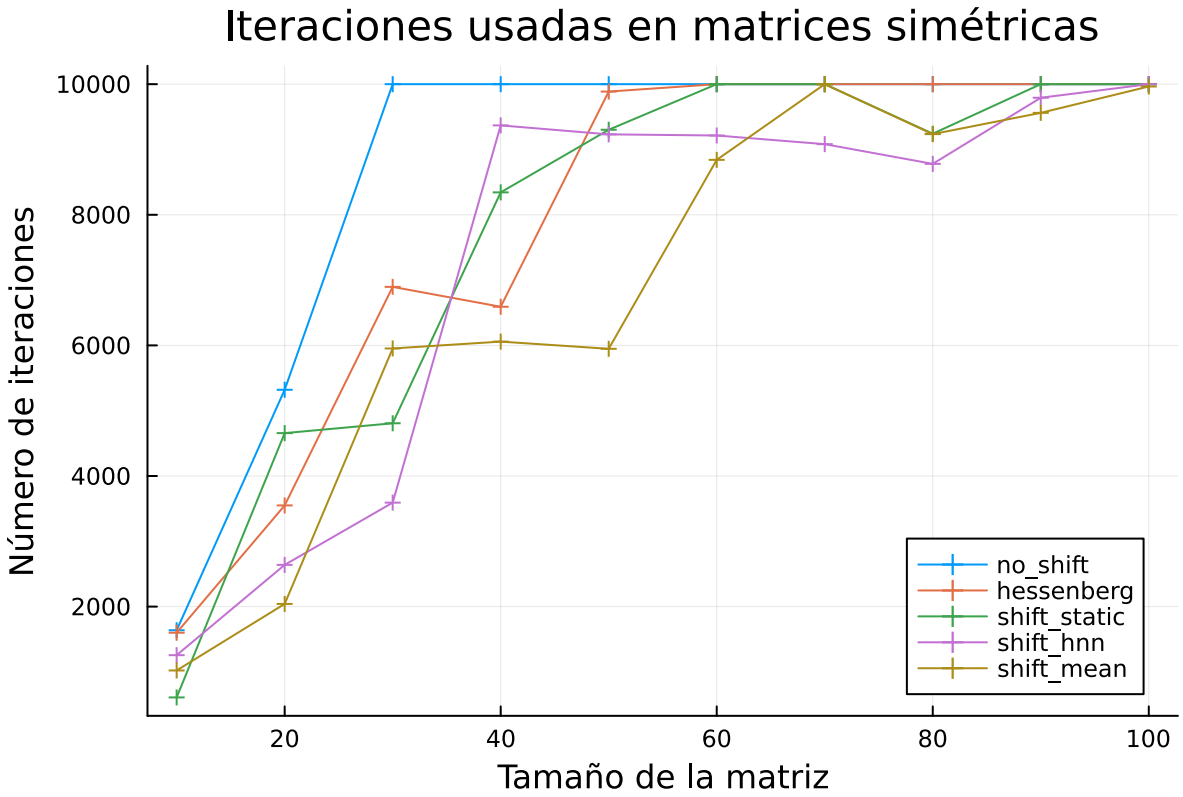
Esto permite analizar la convergencia o falta de ella de los diferentes métodos.

```
mean_per_size (generic function with 1 method)
```

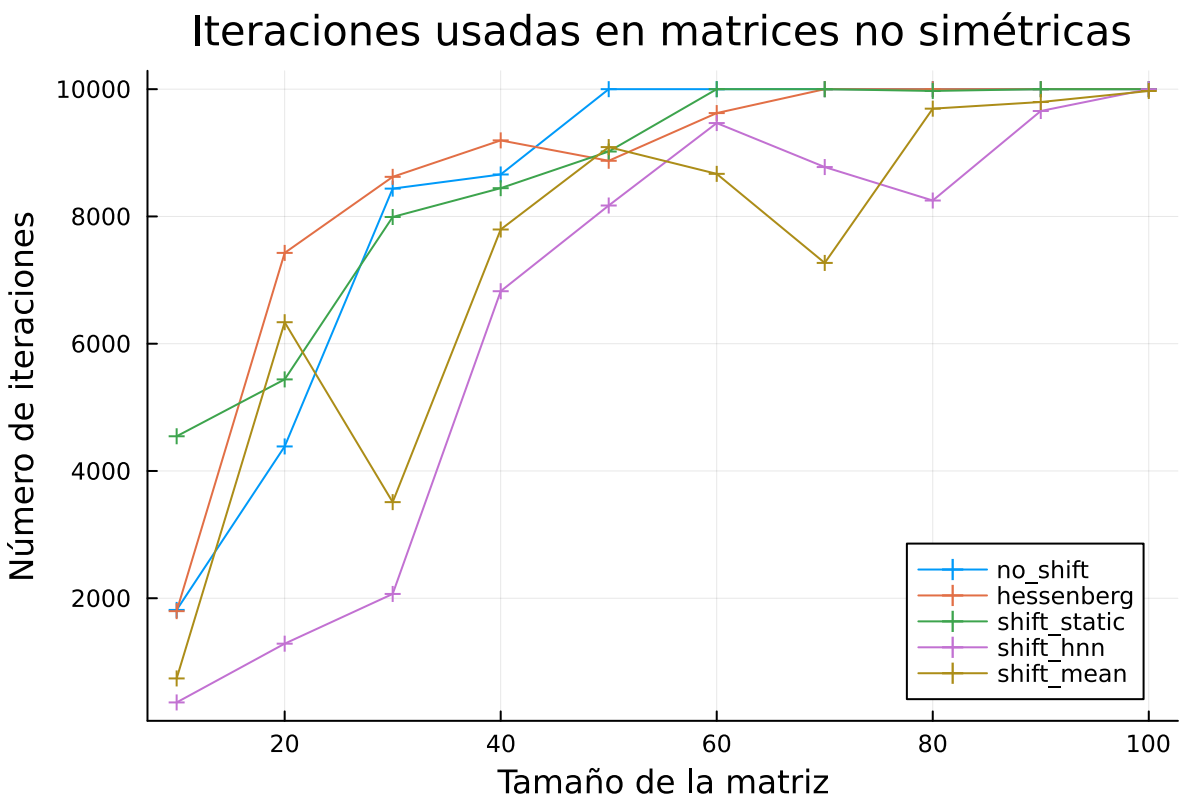
```
1 function mean_per_size(result_matrix::Matrix{<:NamedTuple}, field::Symbol)
2     return [
3                 mean(getfield(r, field)
4                 for r in result_matrix[i, :])
5                 for i in 1:size(result_matrix, 1)
6             ]
7 end
8
```

```
mean_iterations_per_size (generic function with 1 method)
```

```
1 mean_iterations_per_size(result_matrix, _) = mean_per_size(result_matrix,
  :iteration)
```




```
plot_experiment(
    symmetric_experiment,
    mean_iterations_per_size,
    ylabel = "Número de iteraciones",
    title = "Iteraciones usadas en matrices simétricas"
)
```



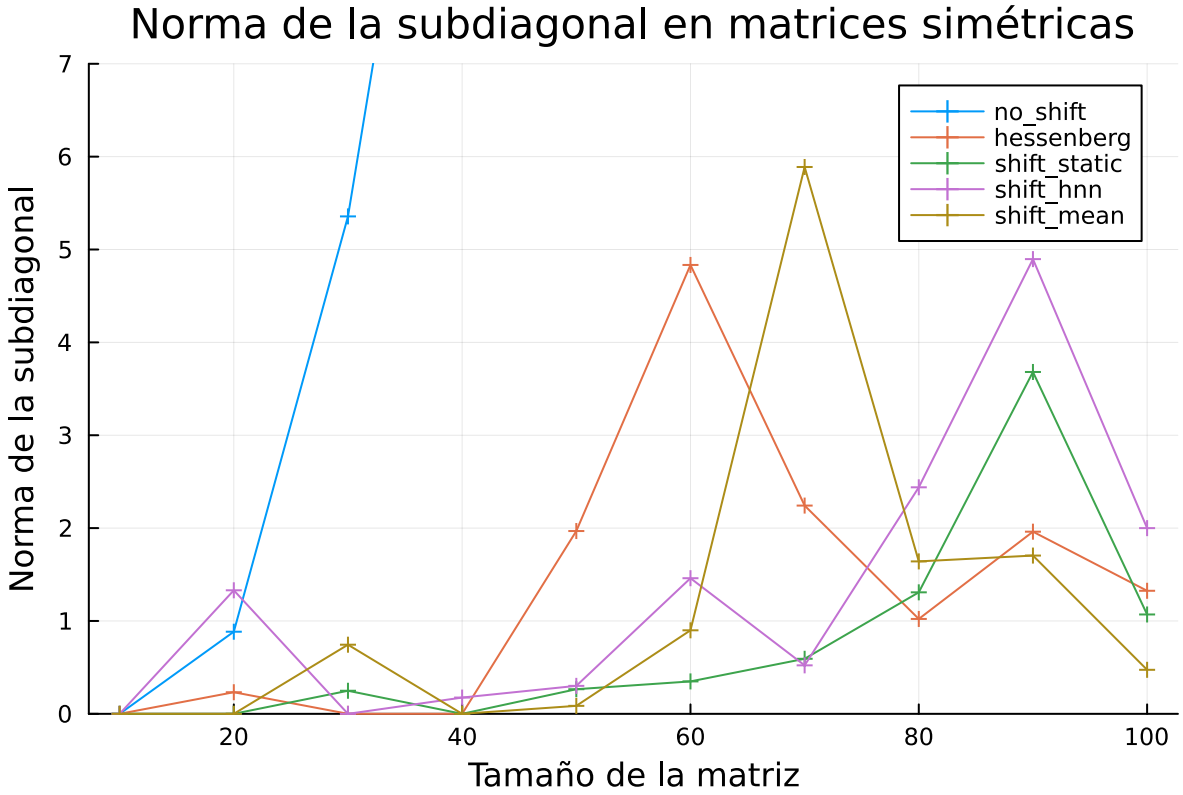

```
plot_experiment(
    nonsymmetric_experiment,
    mean_iterations_per_size,
    ylabel = "Número de iteraciones",
    title = "Iteraciones usadas en matrices no simétricas"
)
```

Norma de la subdiagonal

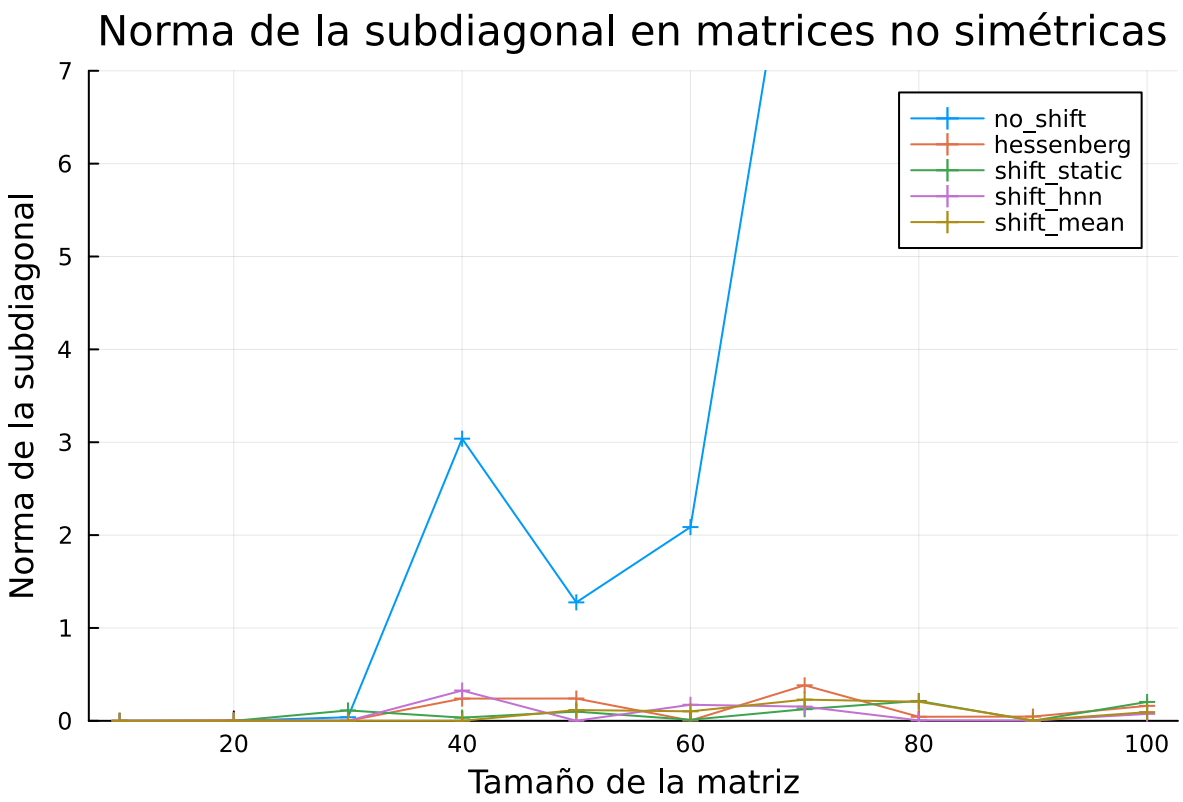
Permite analizar el avance de los algoritmos incluso cuando no han convergido.

```
mean_subdiagonal_norm_per_size (generic function with 1 method)
```

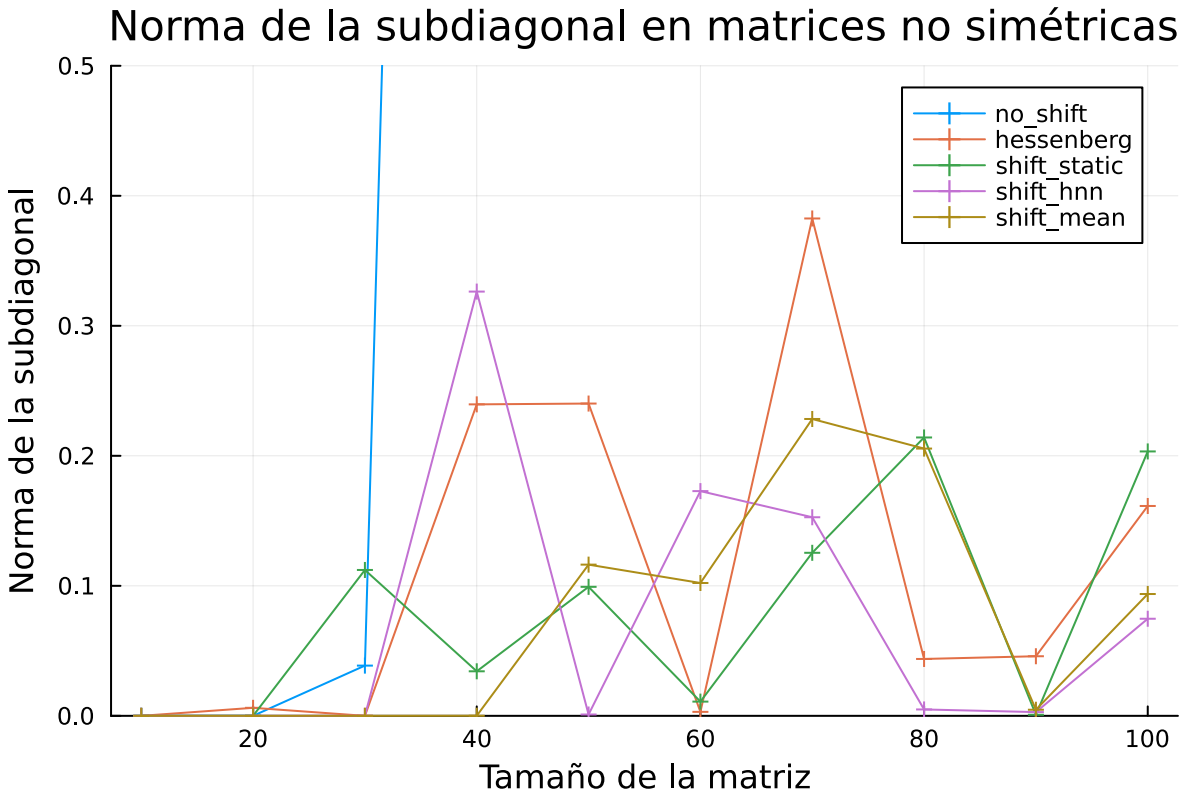
```
function mean_subdiagonal_norm_per_size(result_matrix,_)
    return [
        mean(subdiagonal_norm(r.Ak) for r in result_matrix[i, :])
        for i in 1:size(result_matrix, 1)
    ]
end

```




```
plot_experiment(
    symmetric_experiment,
    mean_subdiagonal_norm_per_size,
    ylabel = "Norma de la subdiagonal",
    title = "Norma de la subdiagonal en matrices simétricas",
    y_max = 7
)
```




```
plot_experiment(
    nonsymmetric_experiment,
    mean_subdiagonal_norm_per_size,
    ylabel = "Norma de la subdiagonal",
    title = "Norma de la subdiagonal en matrices no simétricas",
    y_max = 7
)
```




```
plot_experiment(
    nonsymmetric_experiment,
    mean_subdiagonal_norm_per_size,
    ylabel = "Norma de la subdiagonal",
    title = "Norma de la subdiagonal en matrices no simétricas",
    y_max = .5
)
```

Error espectral absoluto

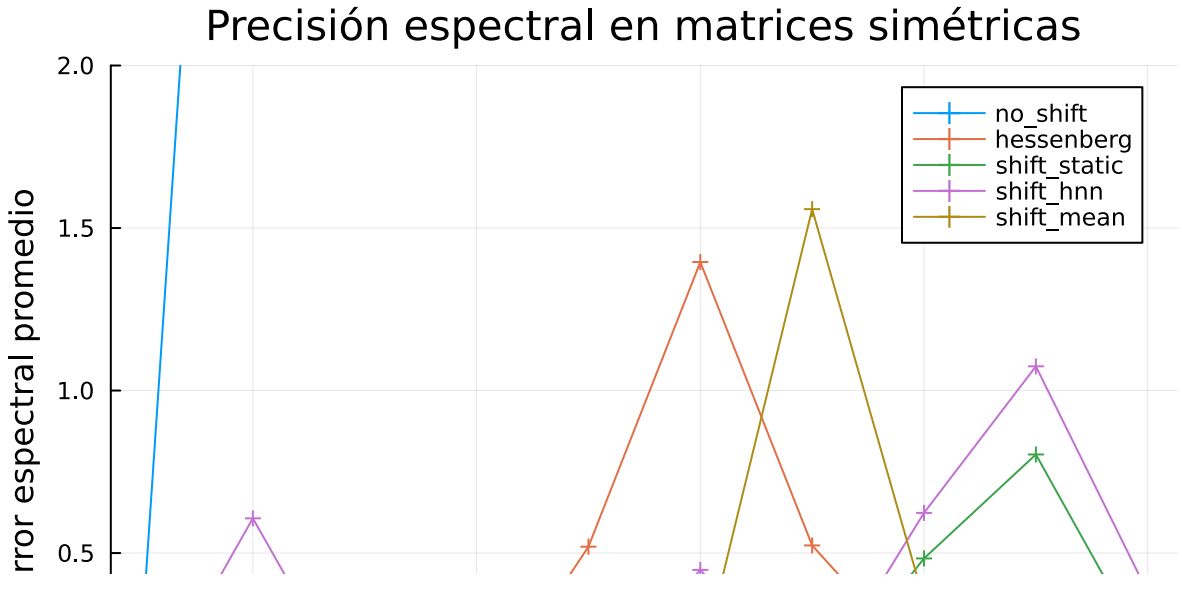
Permite comparar la precisión del resultado del algoritmo: los autovalores de la matriz.

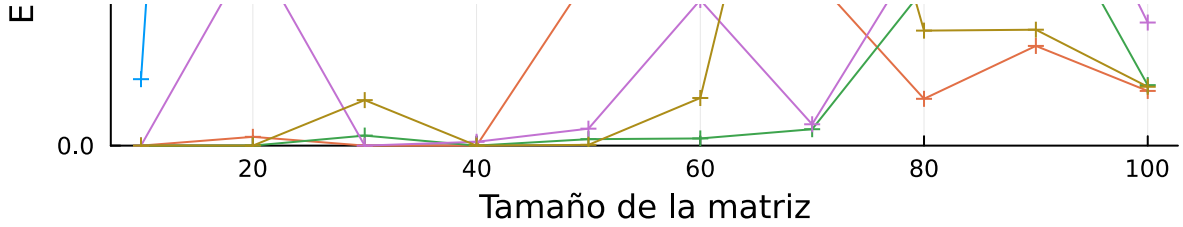
```
spectral_error (generic function with 1 method)
```

```
function spectral_error(result::NamedTuple, A::AbstractMatrix)
    λ_approx = sort(diag(result.Ak))
    λ_exact  = sort(eigvals(A))
    return norm(λ_approx - λ_exact)
end

```

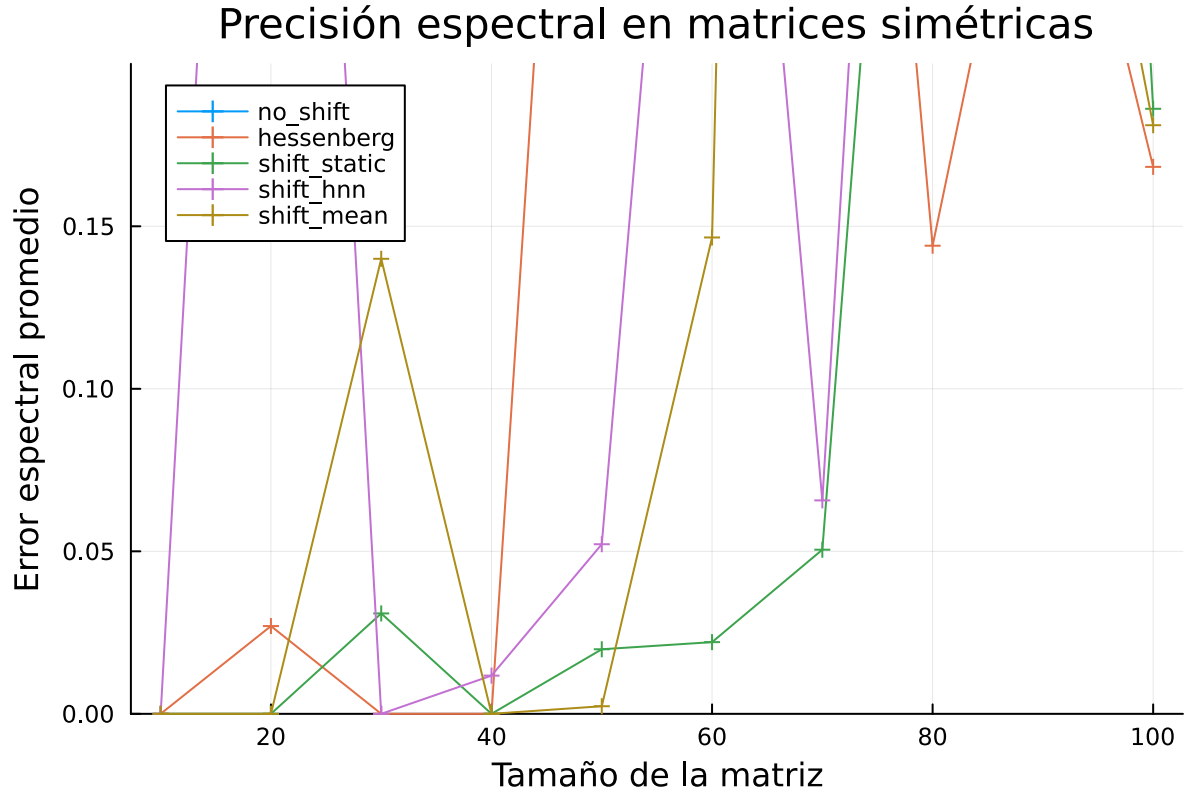
```
mean_spectral_error_per_size (generic function with 1 method)
```

```
function mean_spectral_error_per_size(result_matrix, original_matrix)
    n_rows = size(result_matrix, 1)
    return [
        mean(spectral_error(result_matrix[i, j], original_matrix[i, j]) for j in 1:size(result_matrix, 2))
        for i in 1:n_rows
    ]
end

```

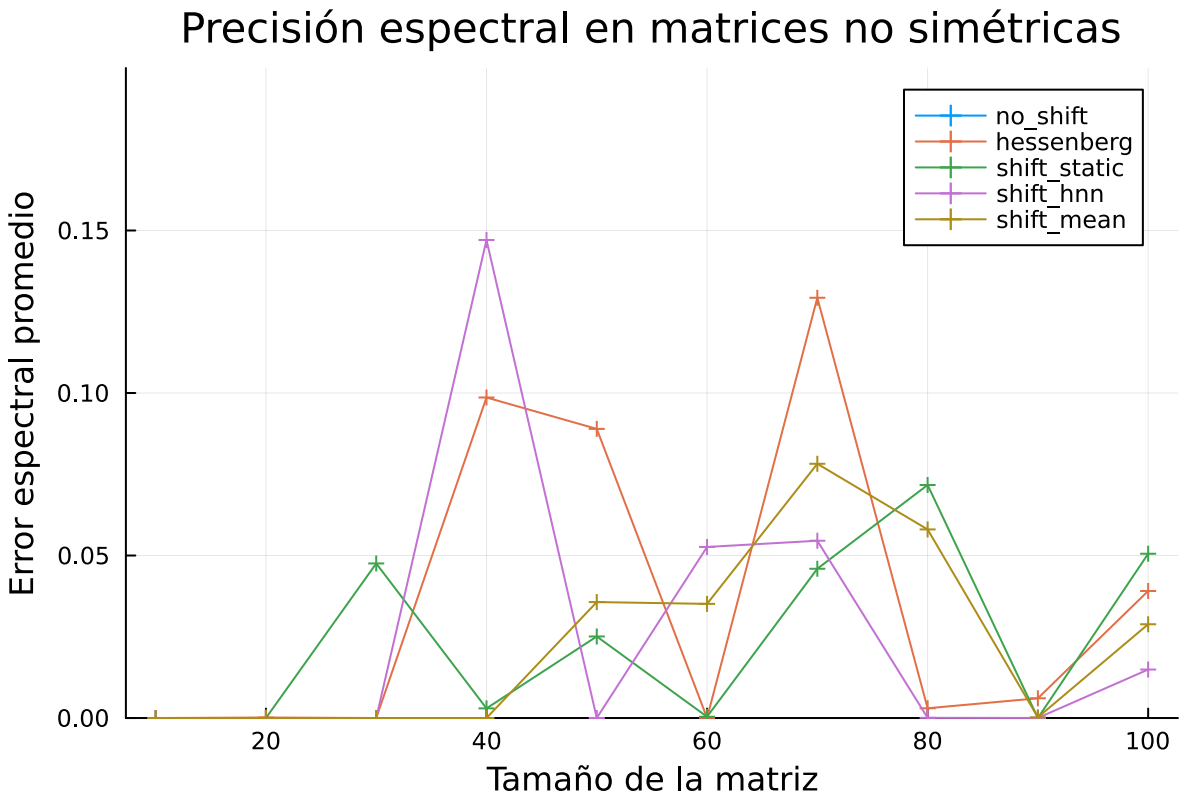

Precisión espectral en matrices simétricas
rror espectral promedio
2.0
1.5
1.0
0.5
no_shift
hessenberg
shift_static
shift_hnn
shift_mean


E
0.0
20
40
60
80
100
Tamaño de la matriz

```
plot_experiment(
    symmetric_experiment,
    mean_spectral_error_per_size,
    ylabel = "Error espectral promedio",
    title = "Precisión espectral en matrices simétricas",
    y_max = 2
)

```




```
plot_experiment(
    symmetric_experiment,
    mean_spectral_error_per_size,
    ylabel = "Error espectral promedio",
    title = "Precisión espectral en matrices simétricas",
    y_max = .2
)

```




```
plot_experiment(
    nonsymmetric_experiment,
    mean_spectral_error_per_size,
    ylabel = "Error espectral promedio",
    title = "Precisión espectral en matrices no simétricas",
    y_max = 0.2
)

```