Tarea 1

Debido a que se debe realizar el mismo proceso varias veces, se construyeron 'workflows' en los que se ejecutan todos los pasos necesarios para obtener los resultados. En cada sección se introducen paulatinamente todas las partes del workflow, ya sea con funciones o con ejemplos.

Objetivo

Analizar experimentalmente la sensibilidad a la precisión numérica y el comportamiento computacional de los algoritmos de Gram-Schmidt (GS) y Gram-Schmidt Modificado (GSM) implementados en Julia, usando diferentes representaciones de punto flotante (Float16, Float32, Float64).

```
begin
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
    using LinearAlgebra
    using Plots
end
```

Algoritmos GS y GSM

Utilizamos la implementación de las notas de clase:

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

```
#Gram - Schmidt clásico
function QRCGS(A)
    sizeA=size(A)
    Q = zeros(sizeA) #(m,n)
    R = zeros(sizeA[2],sizeA[2]) #(n,n)
    for i = 1:sizeA[2]
        for j = 1:i-1
            R[j,i] = Q[:,j]'A[:,i]
        end
        p = A[:,i] - Q[:,1:i-1]*R[1:i-1,i]
        R[i,i]=norm(p)
#        if abs(R[i,i])<0.00000000001
#            println("Rii cercano 0")
#        end
        Q[:,i] = p/R[i,i]
    end
    return Q,R
end
```

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

```
# Gram - Schmidt modificado
function QRMGS(A)
    sizeA=size(A)
    Q = zeros(sizeA) #(m,n)
    R = zeros(sizeA[2],sizeA[2]) #(n,n)
    for i = 1:sizeA[2]
        R[i,i] = norm(A[:,i])
        Q[:,i] = A[:,i]/R[i,i]
        for j = i + 1: sizeA[2]
            R[i,j] = Q[:,i]'A[:,j]
            A[:,j] = A[:,j] - Q[:,i]R[i,j]
        end
    end
    return Q, R
end
```

Familia de matrices

En el ejemplo se genera la familia de matrices aleatorias n=10,50,100,200 y se aplican ambos algoritmos a cada una.

Además, se la función organizeResults, cuya finalidad es separar una lista de tuplas que contienen las matrices Q y R generadas por el algoritmo de factorización. Esta función toma como entrada un arreglo de la forma $[(Q_1, R_1), (Q_2, R_2), ...]$ y devuelve dos arreglos separados: uno con todas las matrices Q y otro con todas las matrices R.

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

```
function organizeResults(array)
    # A partir de un array [(Q₁, R₁), (Q₂, R₂), ...]
    # devuelve un arreglo Q = [Q₁, Q₂, ...] y R = [R₁, R₂, ...].
    Q = []; R = [];
    for i in range(1,size(array)[1])
        push!(Q, array[i][1]);
        push!(R, array[i][2]);
    end
    return Q,R
end
```

Ejemplo

```
sizes_example =   [10, 50, 100, 200]
```

```
sizes_example = [10,50,100,200]
```

Se generan matrices aleatorias de cada tamaño especificado

```
matrices_example =
  [10×10 Matrix{Float64}:                                                , 50×5
    0.207406   0.38328    0.397599  0.568543   …  0.692825   0.128939   0.647838   0.7
```

```
matrices_example = [rand(x,x) for x in sizes_example]
```

Se aplican ambos algoritmos

```
results_QRCGS =
  [(10×10 Matrix{Float64}:
    0.112532   0.200912   0.160695   0.557126   …  0.468056   0.0276034   0.05051
```

```
results_QRCGS = [QRCGS(x) for x in matrices_example]
```

```
results_QRMGS =
  [(10×10 Matrix{Float64}:
    0.112532   0.200912   0.160695   0.557126   …  0.468056   0.0276034   0.05051
```

```
results_QRMGS = [QRMGS(x) for x in matrices_example]
```

Se organizan los resultados en dos arreglos Q y R

```
  ([10×10 Matrix{Float64}:
    0.112532   0.200912   0.160695   0.557126   …  0.468056   0.0276034   0.05051
```

```
Q_QRCGS,R_QRCGS = organizeResults(results_QRCGS)
```

Mediciones

Para cada tipo de precisión (Float16, Float32, Float64), medir:

- **Tiempo de ejecución** de cada algoritmo.
- **Error de ortogonalidad:** ‖Q^TQ−I‖, \| Q^T Q - I \|,
- **Residuo de la factorización QR:** ‖A−QR‖ o

Errores

Medición del residuo absoluto de la factorización y residuo de la ortogonalización.

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

```
function defineErrors(matrices,Q,R)
    absError = []
    ortError = []
    for i in range(1,size(matrices)[1])
        append!(absError, opnorm(matrices[i]-Q[i]*R[i]))
        append!(ortError, opnorm(Q[i]'Q[i]-UniformScaling(1)))
    end
    return Dict("absoluteError"=>absError,"ortogonalizationError"=>ortError)
end
```

Tiempo

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

```
function time_QRCGS(matrices)
    return [@elapsed QRCGS(x) for x in matrices]
end
```

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

```
function time_QRMGS(matrices)
    return [@elapsed QRMGS(x) for x in matrices]
end
```

Ejemplo

```
example_error =
  Dict("absoluteError" ⇒ [4.28418, 24.4296, 49.5747, 99.1145], "ortogonalizationError"
```

```
example_error = defineErrors(matrices_Float16,Q_QRCGS,R_QRCGS)
```

```
[1.5235e-5, 0.000255863, 0.00245213, 0.0149777]
```

```
time_QRCGS(matrices_Float16)
```

```
[1.7272e-5, 0.00105162, 0.00853761, 0.0658258]
```

```
time_QRMGS(matrices_Float16)
```

Workflow

Se organizan los pasos anteriores en una función

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

```
function workflow_QRCGS(matrices)
    results = [QRCGS(x) for x in matrices]
    Q,R = organizeResults(results)
    return defineErrors(matrices,Q,R), time_QRCGS(matrices)
end
```

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

```
function workflow_QRMGS(matrices)
    results = [QRMGS(x) for x in matrices]
    Q,R = organizeResults(results)
    return defineErrors(matrices,Q,R), time_QRMGS(matrices)
end
```

Instancias

Utilizando el código desarrollado, se ejecutó todos los ejemplos requeridos cubriendo las seis combinaciones posibles entre los tres niveles de precisión (Float16, Float32, Float64) y los dos algoritmos de factorización (Gram-Schmidt y Gram-Schmidt Modificado).

Para cada combinación se calcularon el error de ortogonalidad, el error absoluto de factorización y el tiempo de ejecución correspondientes.

```
sizes =  [10, 50, 100, 200]
```

```
sizes = [10,50,100,200]
```

```
(Dict("absoluteError" => [4.89718, 24.283, 49.3529, 99.1103], "ortogonalizationError"
```

```
begin
    matrices_Float16 = [rand(Float16,x,x) for x in sizes]
    error_16_QRCGS, time_16_QRCGS =workflow_QRCGS(matrices_Float16)
    error_16_QRMGS, time_16_QRMGS =workflow_QRMGS(matrices_Float16)
end
```

```
(Dict("absoluteError" => [4.82057, 24.4449, 48.9686, 99.0767], "ortogonalizationError"
```

```
begin
    matrices_Float32 = [rand(Float32,x,x) for x in sizes]
    error_32_QRCGS, time_32_QRCGS =workflow_QRCGS(matrices_Float32)
    error_32_QRMGS, time_32_QRMGS =workflow_QRMGS(matrices_Float32)
end
```

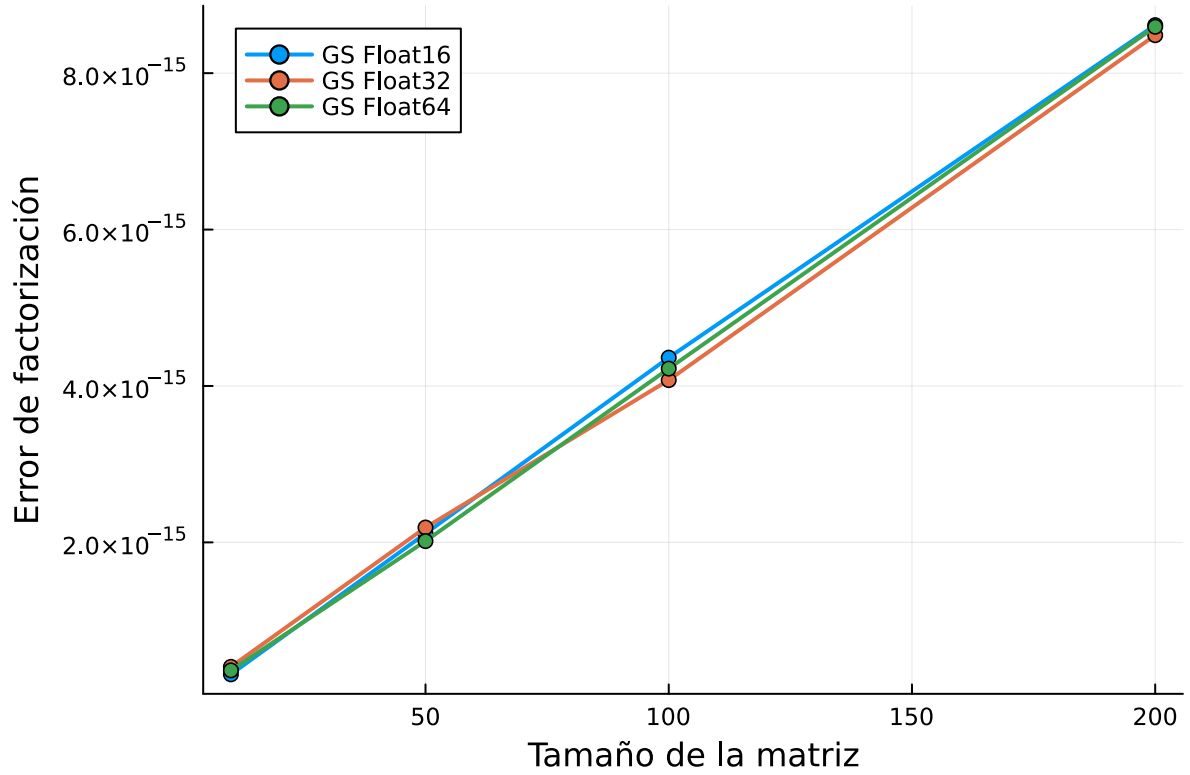
```
(Dict("absoluteError" => [4.95132, 24.5287, 49.555, 99.3157], "ortogonalizationError"
```

```
begin
    matrices_Float64 = [rand(Float64,x,x) for x in sizes]
    error_64_QRCGS, time_64_QRCGS =workflow_QRCGS(matrices_Float64)
    error_64_QRMGS, time_64_QRMGS =workflow_QRMGS(matrices_Float64)
end
```

Gráfica de resultados

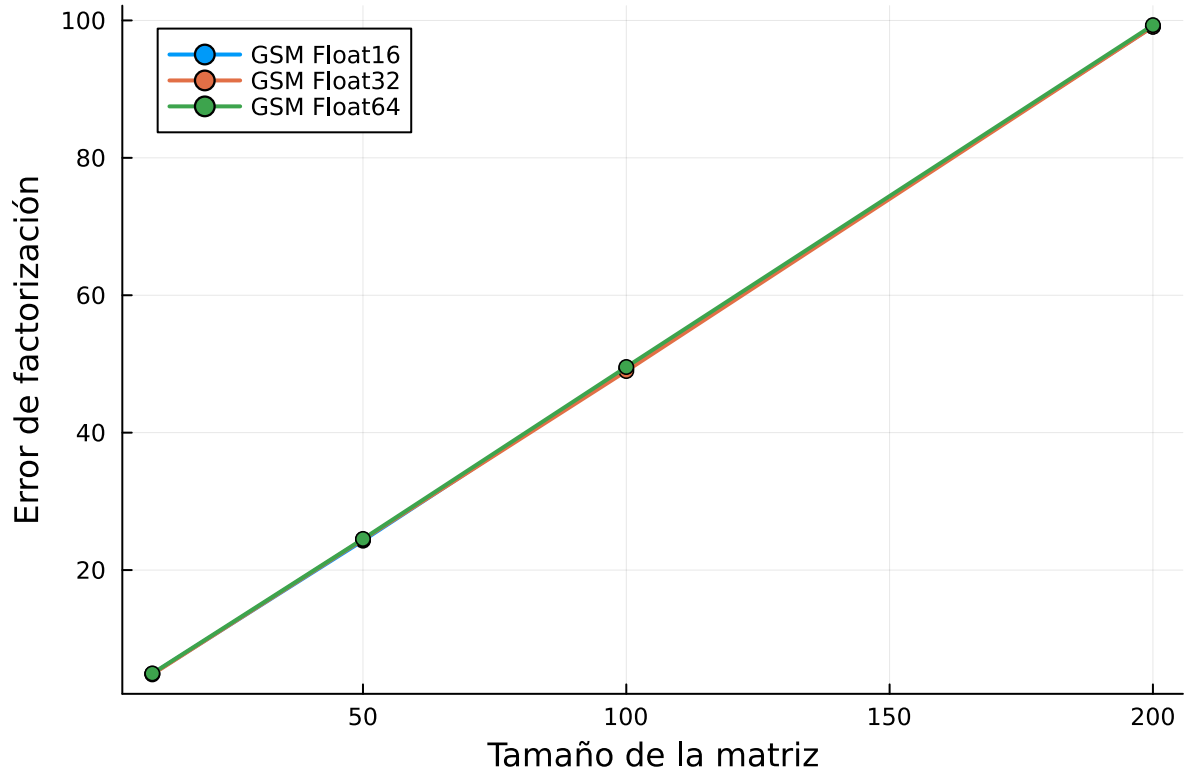
Error de factorización

Algoritmo clásico



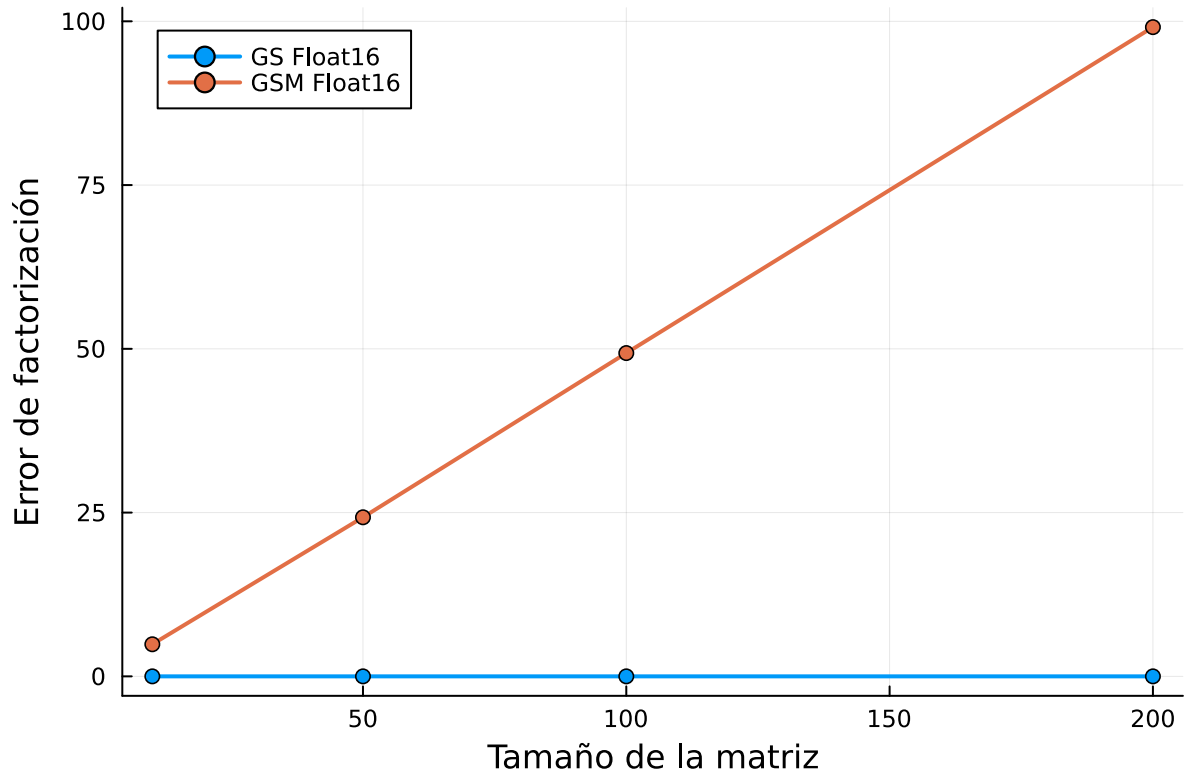

```
begin
    plot(sizes, error_16_QRCGS["absoluteError"], label="GS Float16", lw=2, marker=:circle)
    plot!(sizes, error_32_QRCGS["absoluteError"], label="GS Float32", lw=2, marker=:circle)
    plot!(sizes, error_64_QRCGS["absoluteError"], label="GS Float64", lw=2, marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de factorización")
end
```

Algoritmo modificado



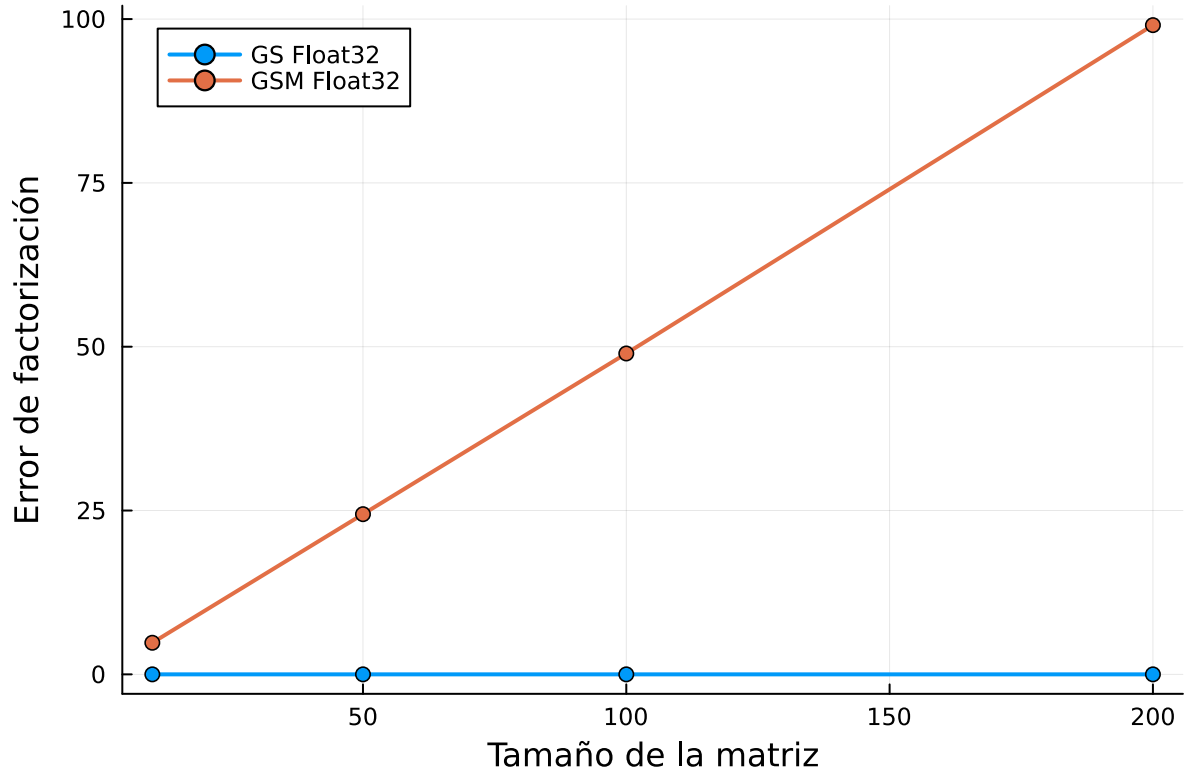

```
begin
    plot(sizes, error_16_QRMGS["absoluteError"], label="GSM Float16", lw=2,
    marker=:circle)
    plot!(sizes, error_32_QRMGS["absoluteError"], label="GSM Float32", lw=2,
    marker=:circle)
    plot!(sizes, error_64_QRMGS["absoluteError"], label="GSM Float64", lw=2,
    marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de factorización")
end
```

Precisión Float16



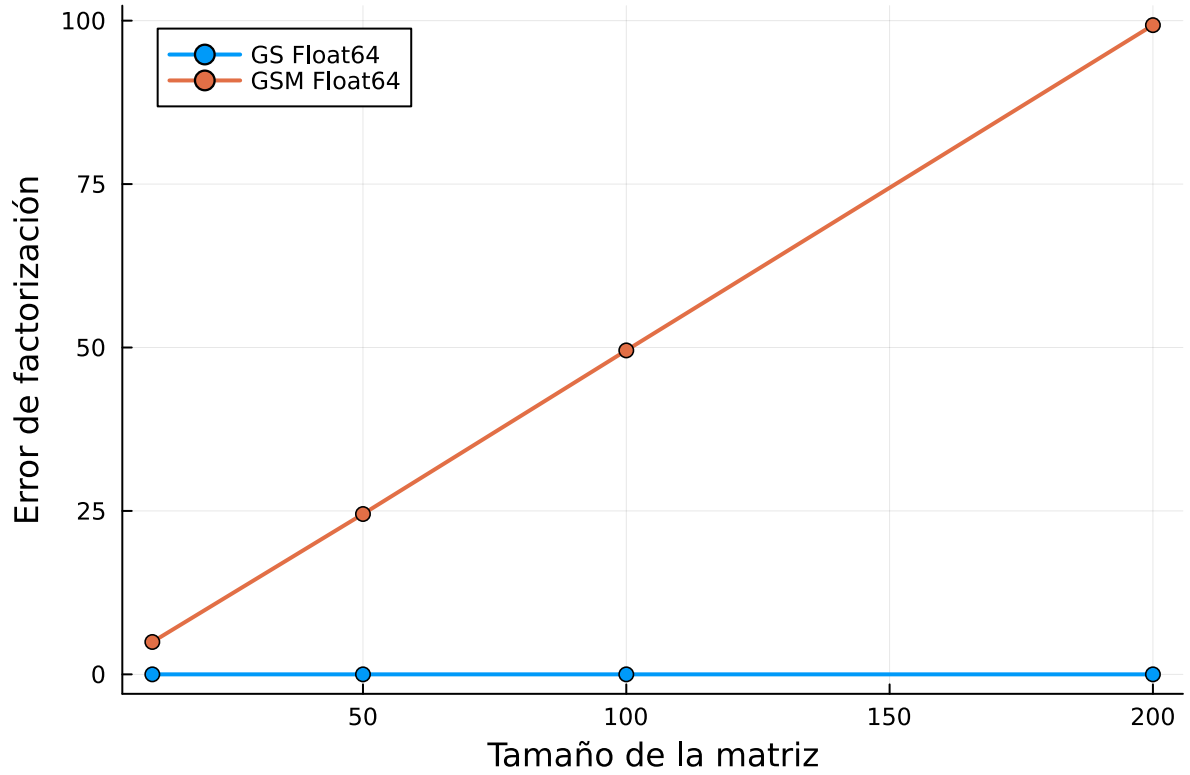

```
begin
    plot(sizes, error_16_QRCGS["absoluteError"], label="GS Float16", lw=2,
    marker=:circle)
    plot!(sizes, error_16_QRMGS["absoluteError"], label="GSM Float16", lw=2,
    marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de factorización")
end
```

Precisión Float32




```
begin
    plot(sizes, error_32_QRCGS["absoluteError"], label="GS Float32", lw=2,
    marker=:circle)
    plot!(sizes, error_32_QRMGS["absoluteError"], label="GSM Float32", lw=2,
    marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de factorización")
end
```

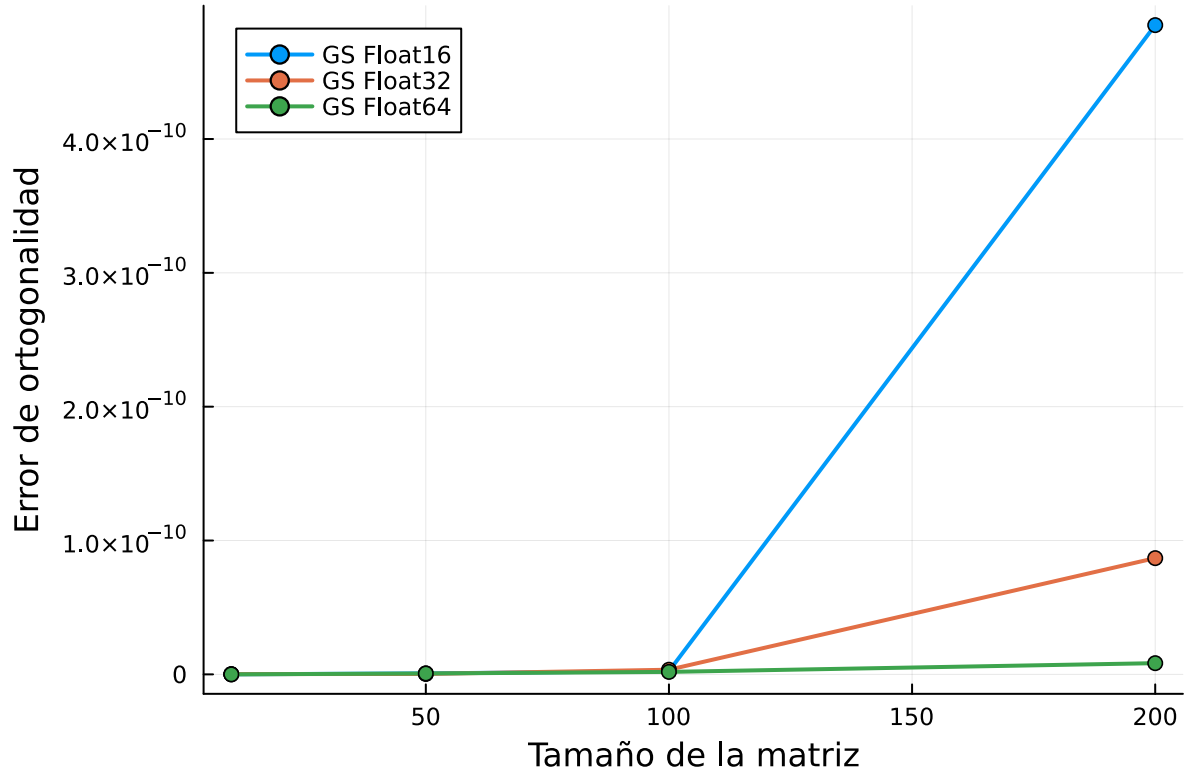
Precisión Float64




```
begin
    plot(sizes, error_64_QRCGS["absoluteError"], label="GS Float64", lw=2,
    marker=:circle)
    plot!(sizes, error_64_QRMGS["absoluteError"], label="GSM Float64", lw=2,
    marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de factorización")
end
```

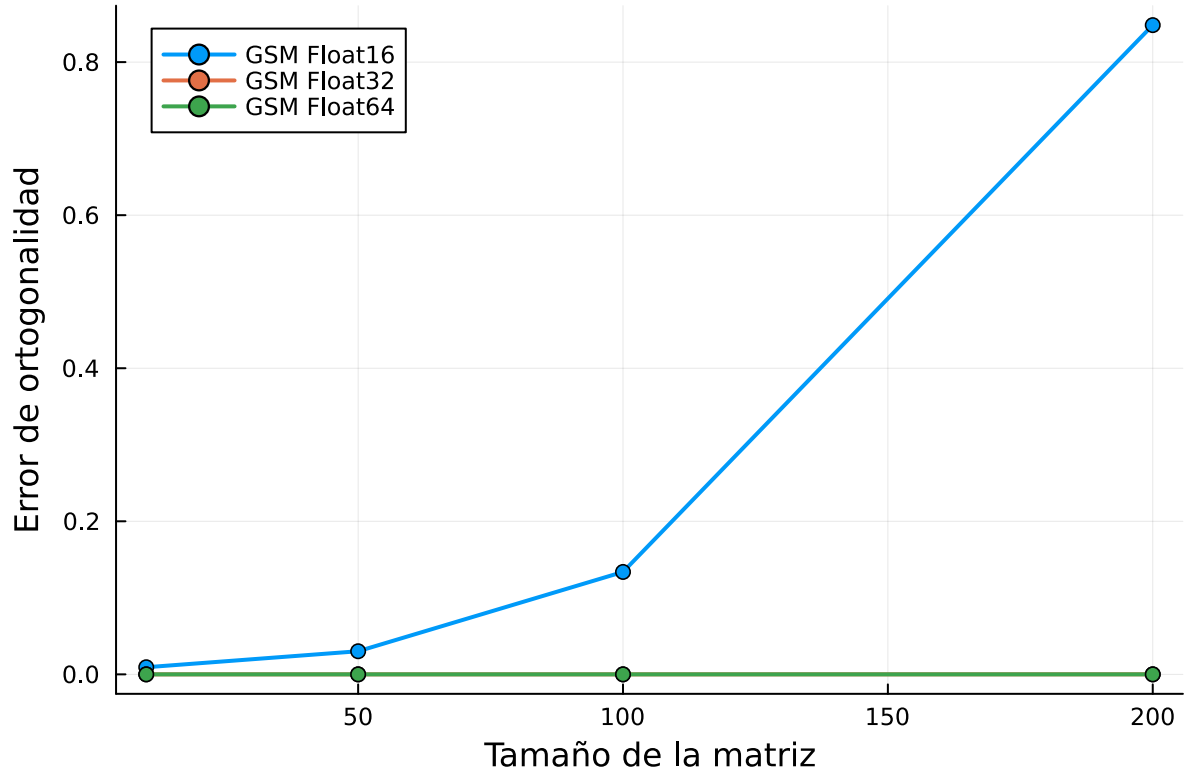
Error de ortogonalidad

Algoritmo clásico



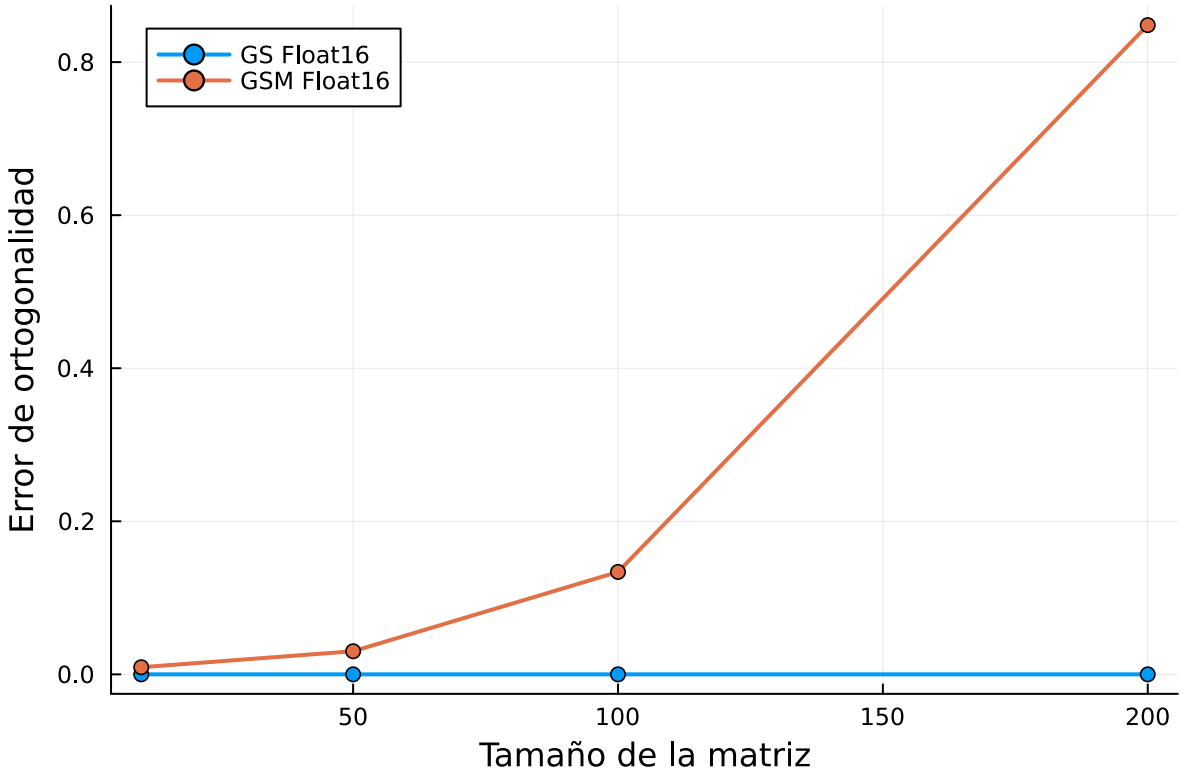

```
1 begin
2     plot(sizes, error_16_QRCGS["ortogonalizationError"], label="GS Float16",
      lw=2, marker=:circle)
3     plot!(sizes, error_32_QRCGS["ortogonalizationError"], label="GS Float32",
      lw=2, marker=:circle)
4     plot!(sizes, error_64_QRCGS["ortogonalizationError"], label="GS Float64",
      lw=2, marker=:circle)
5     xlabel!("Tamaño de la matriz")
6     ylabel!("Error de ortogonalidad")
7 end
```

Algoritmo modificado



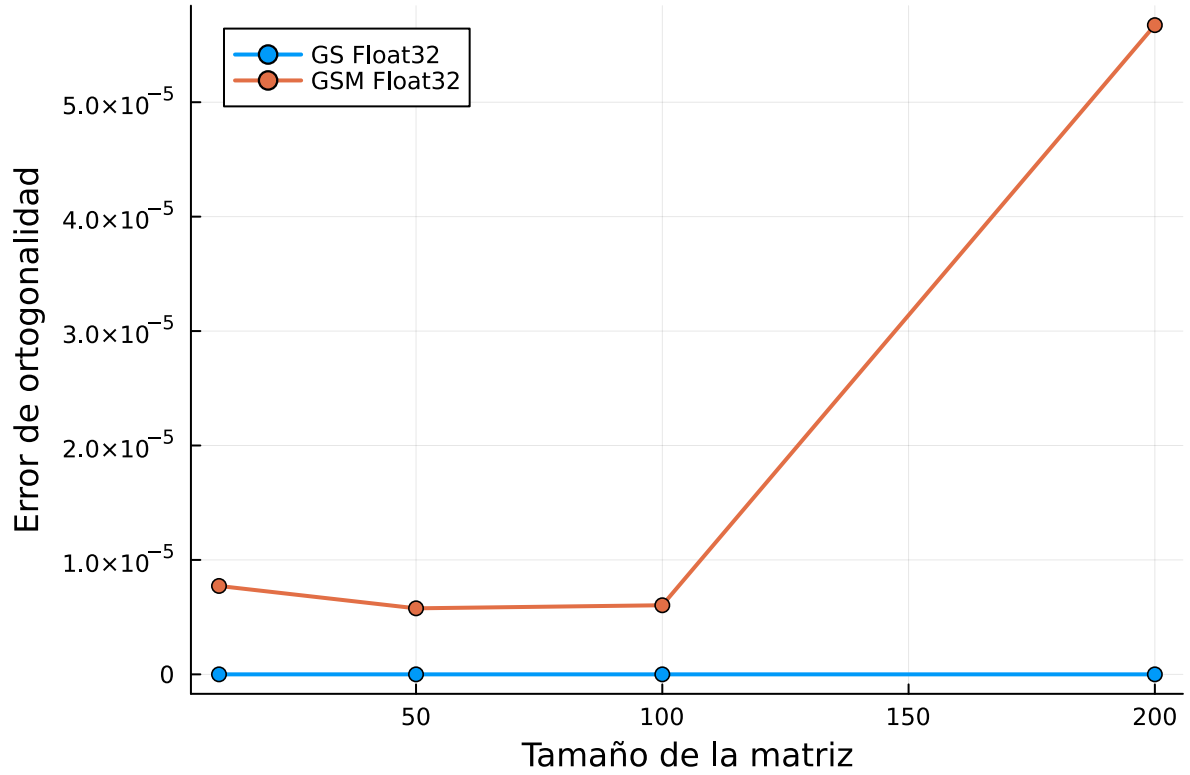

```
begin
    plot(sizes, error_16_QRMGS["ortogonalizationError"], label="GSM Float16", lw=2, marker=:circle)
    plot!(sizes, error_32_QRMGS["ortogonalizationError"], label="GSM Float32", lw=2, marker=:circle)
    plot!(sizes, error_64_QRMGS["ortogonalizationError"], label="GSM Float64", lw=2, marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de ortogonalidad")
end
```

Precisión Float16



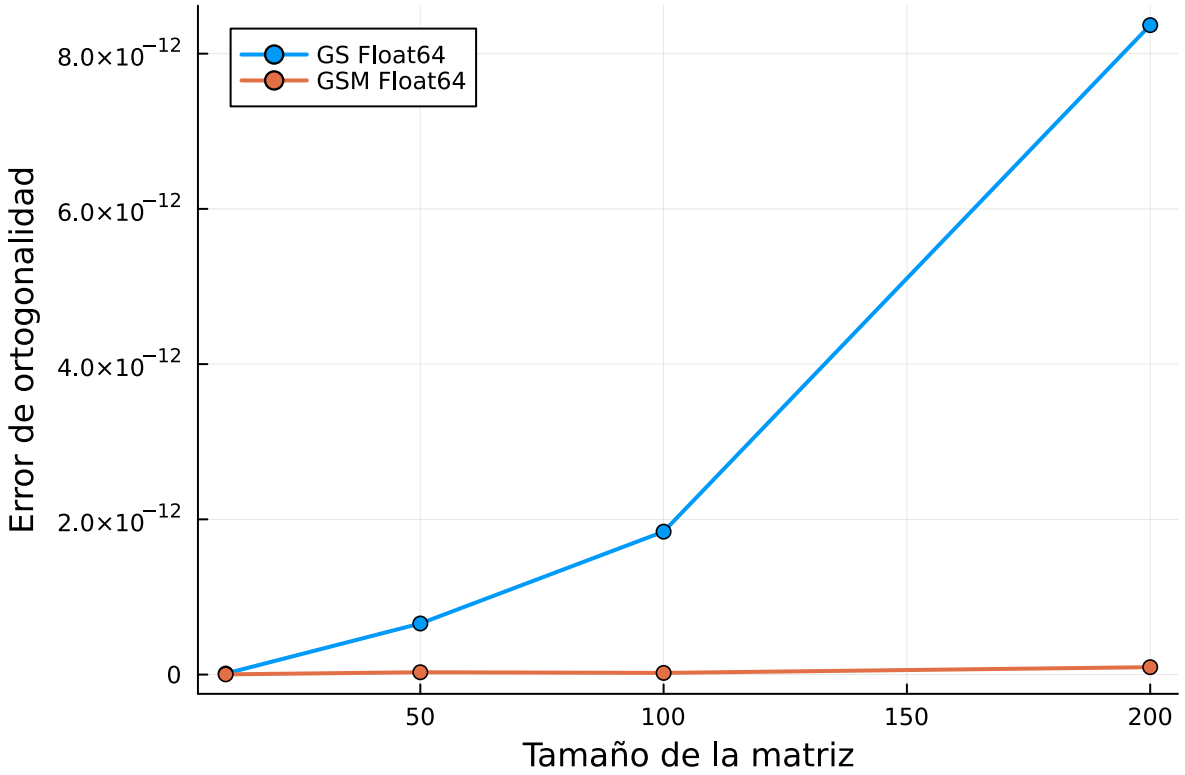

```
begin
    plot(sizes, error_16_QRCGS["ortogonalizationError"], label="GS Float16", lw=2, marker=:circle)
    plot!(sizes, error_16_QRMGS["ortogonalizationError"], label="GSM Float16", lw=2, marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de ortogonalidad")
end
```

Precisión Float32



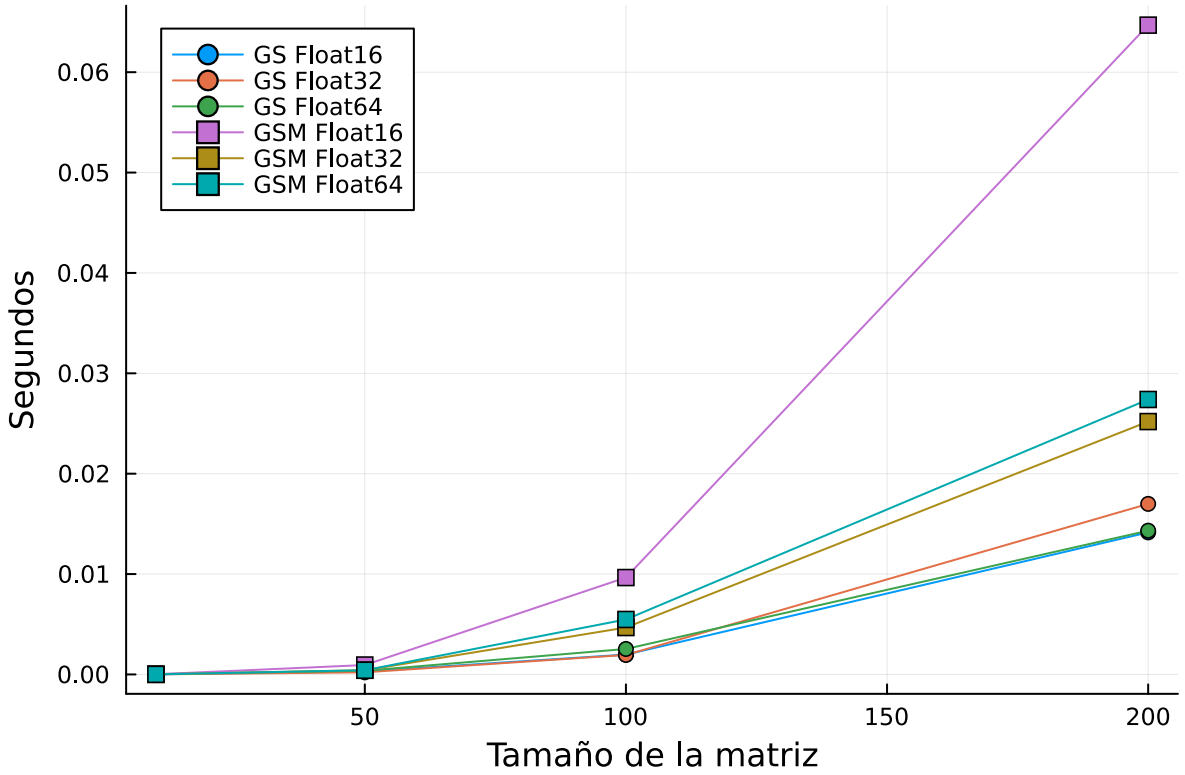

```
begin
    plot(sizes, error_32_QRCGS["ortogonalizationError"], label="GS Float32", lw=2, marker=:circle)
    plot!(sizes, error_32_QRMGS["ortogonalizationError"], label="GSM Float32", lw=2, marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de ortogonalidad")
end
```

Precisión Float64




```
begin
    plot(sizes, error_64_QRCGS["ortogonalizationError"], label="GS Float64", lw=2, marker=:circle)
    plot!(sizes, error_64_QRMGS["ortogonalizationError"], label="GSM Float64", lw=2, marker=:circle)
    xlabel!("Tamaño de la matriz")
    ylabel!("Error de ortogonalidad")
end
```

Velocidad




```
begin
    plot(sizes, time_16_QRCGS, label="GS Float16", marker=:circle)
    plot!(sizes, time_32_QRCGS, label="GS Float32", marker=:circle)
    plot!(sizes, time_64_QRCGS, label="GS Float64", marker=:circle)
    plot!(sizes, time_16_QRMGS, label="GSM Float16", marker=:square)
    plot!(sizes, time_32_QRMGS, label="GSM Float32", marker=:square)
    plot!(sizes, time_64_QRMGS, label="GSM Float64", marker=:square)
    xlabel!("Tamaño de la matriz")
    ylabel!("Segundos")
end
```

Análisis de gráficas

Este análisis está basado en las 3 diferentes ejecuciones realizadas de cada algoritmo. Por lo cual, sus conclusiones no son totalmente fiables.

> ¿Cuál de los dos algoritmos es más estable numéricamente?

De acuerdo a las gráficas, podemos decir que el algoritmo clásico es el más estable numéricamente. Esto debido a que tiene un menor error de factorización y de ortogonalidad en todas menos una de las gráficas comparativas (la comparación de error de ortogonalidad para Float64).

> ¿Cómo afecta la precisión (Float16, Float32, Float64) a cada algoritmo?

La precisión no afecta el error de factorización de los algoritmos ni el error de ortogonalidad del algoritmo clásico. En cambio, sí afecta claramente el error de ortogonalidad del algoritmo modificado.

¿Cuál es más rápido? ¿A partir de qué tamaño?

El más rápido es el algoritmo clásico, desde n=100.

Reflexión

Durante el desarrollo de esta tarea enfrenté diversas dificultades, principalmente relacionadas con la manipulación de estructuras de datos en Julia, como listas de tuplas, arrays de matrices y diferencias sutiles en el tratamiento de filas y columnas. También surgieron desafíos al aplicar funciones sobre arreglos de forma vectorizada sin perder la estructura de los datos.

A pesar de estas dificultades, el proceso permitió afianzar mi comprensión sobre la implementación de algoritmos numéricos, el manejo de precisión numérica, y el diseño de flujos de trabajo reproducibles en Julia. Además, adquirí mayor familiaridad con herramientas para medir errores, tiempos de ejecución y organizar resultados de forma sistemática. En conjunto, la experiencia fortaleció tanto mis habilidades de programación como mi intuición numérica.

Fuentes externas

Durante el desarrollo de este proyecto, consulté diversas fuentes externas para comprender y aplicar correctamente conceptos y estructuras del lenguaje de programación Julia. Las principales fuentes utilizadas fueron:

- Notas de clase: De las cuales obtuve las funciones QRCGS y QRMGS.
- Documentación oficial de Julia: Utilicé la documentación oficial para entender y aplicar correctamente las estructuras de datos como arrays y diccionarios. En particular, consulté las secciones sobre arrays y colecciones y estructuras de datos .
- Foros de discusión y comunidades en línea: Recurrí a plataformas como Stack Overflow para resolver dudas específicas sobre la implementación y manipulación de arrays y diccionarios en Julia. Por ejemplo, consulté discusiones sobre cómo crear un array de unos y cómo iterar sobre un diccionario .

Declaración del uso de inteligencia artificial

En el desarrollo de este proyecto utilicé el modelo de lenguaje ChatGPT de OpenAI como herramienta de apoyo para resolver dudas conceptuales y técnicas relacionadas con la implementación y análisis de algoritmos numéricos en el lenguaje Julia.

Los usos principales de la IA incluyeron:

- Consulta sobre sintaxis y estructuras idiomáticas en Julia para la creación de arrays derivados. Prompt utilizado: “¿Cómo puedo definir en Julia un nuevo array a partir de uno existente?” y luego: “Cómo puedo definir dos arrays a partir de uno original? Para que sea equivalente a ‘a,b = func()’” y: “Quiero formar dos nuevos arrays Y y Z a partir de X tal que y[0], z[0] = func(x[0])”
- Ayuda para aplicar funciones sobre filas o columnas de una matriz, asegurando el comportamiento esperado en contextos donde los datos no eran escalar por escalar. Prompt utilizado: “Tengo un problema, el argumento de la función func() está siendo

tomado como un vector en lugar de una matriz”
- Generación de código para graficar errores de ortogonalidad y factorización utilizando Plots.jl. Prompt utilizado: “Cómo podría presentarlos en una gráfica?”
- Asesoría sobre cómo medir la estabilidad numérica de un algoritmo, incluyendo métricas concretas como el error de ortogonalidad y de factorización. Prompt utilizado: “Cómo puedo medir la estabilidad numérica del algoritmo?”
- Captura del tiempo de ejecución de funciones junto con sus resultados. Prompt utilizado: “Usando @time en julia, cómo puedo guardar el tiempo demorado y el resultado de la función”
- Redacción.

Todas las decisiones de implementación, validación de resultados y análisis de los datos obtenidos fueron realizados personalmente. La inteligencia artificial se utilizó como herramienta complementaria de consulta, y no sustituyó mi juicio académico ni la autoría del trabajo.