# Numerical Analysis and Optimization Homework Project 1

*Marco Riggirello, Francesco Vaselli*

This assignment is done using the Julia programming language, a high-level, high-performance language for technical computing. To easy the task, we load some Julia packages:

- **LinearAlgebra.jl**: the standard library for LinearAlgebra routines and utilites, leveragin BLAS subroutines;
- **Random.jl**: for generating random numbers;
- **Statistics.jl**: for computing statistics of samples;
- **Plots.jl**: for plotting our results;
- **BenchmarkTools.jl**: to provide benchmarking of our implementations.

```julia
using LinearAlgebra, Plots, Statistics, Random, BenchmarkTools
```

## Problem 1

### Task 1

Here we define the function `lufact()` that takes as input a square matrix and computes the non-pivoted LU factorization and its growth factor.

The $L$ and $U$ matrices are computed via Gaussian elimination, as presented in the class, while the growth factor $\gamma$ is defined as the ratio of the largest entry in the matrix $G \equiv |L||U|$ and the largest entry in the matrix $|A|$ (absolute values are element-wise).

In our implementation, Gaussian elimination is carried on by storing both $L$ and $U$ in a single wokring matrix $A_k$. In the last iteration $n$, the upper triangular part of $A_n$ represents the nonzero elements of $U$ while the lower part represent the $L$ matrix, diagonal excluded. The $A_k$ update is done using the slicing syntax of Julia's arrays.

We made our implementation more robust in various ways:

- in the first place, the Julia idiomatic type declaration (`function lufact(A::AbstractMatrix{T}) where T <: Union{Real, Complex}`) allowed us to know how to correcly initialize $A_k$ and $\gamma$ based on the matrix type. Specifically, for every subtype of `Real`, we initialize the copy to `Float64`, so that integers matrix factorization will not fail due to the creation of floating-point values in an integer matrix;
- furthermore, a preliminary check on the matrix squareness is performed;
- At every $A_k$ update, we check that the pivot is greater than the machine epsilon to ensure a sensible result;
- Ath the end, we also check for the presence of `NaN`s in $A_k$, sign of numerical instability in this non-pivoted variant of the LU factorization.

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

```julia
function lufact(A::AbstractMatrix{T}) where T <: Union{Real, Complex}
    m, n = size(A)
    # Ensures the matrix is square.
    if m != n
        throw(ArgumentError("Matrix is not square."))
    end

    # Creates a copy of A
    U = T <: Real ? Float64 : ComplexF64

    Aₖ = map(U, A)
    γ = zero(U)

    # Performs the LU factorization.
    for k in 1:n-1
        pivot = Aₖ[k, k]
        # Sanity check
        if abs(pivot) <= eps(Float64)
            throw(DomainError("Matrix is ill-conditioned."))
        end
        # Compute i-th column of L
        Aₖ[k+1:n,k] = Aₖ[k+1:n,k] / pivot
        # Update Aₖ
        Aₖ[k+1:n,k+1:n] -= Aₖ[k+1:n,k] * (Aₖ[k,k+1:n])'

    end
    # check if we ended up with NaNs
    if any(isnan, Aₖ)
        throw(DomainError("Factorization is numerically unstable for this
        matrix."))
    end
    # Constructs L and U from the updated matrix Aₖ.
    L = UnitLowerTriangular(Aₖ)
    U = UpperTriangular(Aₖ)

    # Computes the growth factor γ.
    G = abs.(L) * abs.(U)
    γ = maximum(G) / maximum(abs.(A))

    return L, U, γ
end
```

## A note on growth factor

Ideally, the growth factor should be close to 1. This indicates that the LU decomposition did not significantly increase the magnitude of the matrix's elements, suggesting a stable decomposition. It's important to note that a low growth factor does not guarantee a good decomposition. It's just one of several indicators of the quality and stability of the decomposition.

## Task 2 and 3

Now we want to test the performances of this algorithm over various matrix types. Before we proceed, we define a bunch of utility functions:

- introdurre in elenco?

RICORDIAMOCI DI COMMENTARE I RISULTATI SULLE VARIE MATRICI

```
md"""
Now we want to test the performances of this algorithm over various matrix types. Before we proceed, we define a bunch of utility functions:
- introdurre in elenco?

RICORDIAMOCI DI COMMENTARE I RISULTATI SULLE VARIE MATRICI
"""
```

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

```
function relative_backward_error(A, L, U)
    return opnorm(A - L * U, Inf)/opnorm(A, Inf)
end
```

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

```
function print_summary(g, b, f, t)
    println("Failure rate: $(100*f/t) %")
    # add check if g, b are empty
    if length(g) == 0 || length(b) == 0
        println("No data to show.")
        return
    end
    println("GROWTH FACTOR")
    println("Min:    $(minimum(g))")
    println("Max:    $(maximum(g))")
    println("Mean:   $(mean(g))")
    println("Median: $(median(g))")
    println("StdDev: $(std(g))")
    println("RELATIVE BACKWARD ERROR")
    println("Min:    $(minimum(b))")
    println("Max:    $(maximum(b))")
    println("Mean:   $(mean(b))")
    println("Median: $(median(b))")
    println("StdDev: $(std(b))")
end
```

```
plot_summary (generic function with 2 methods)
```

```julia
function plot_summary(g, b, pq=1.)
    # Adjust the binning for histograms if necessary to make the x-axis less crowded
    # Adjust the size of the plot or the labels to prevent squeezing
    hg = histogram(g[g .<= quantile(g, pq)], xlabel="Growth factor", ylabel="Frequency",
                   legend=false, xrotation=45, bins=30) # Rotate x-axis labels and set bins
    hb = histogram(b[b .<= quantile(g, pq)], xlabel="Relative backward error", ylabel="Frequency",
                   legend=false, xrotation=45, bins=30) # Rotate x-axis labels and set bins

    # Combine the two histograms into one plot without a legend
    plot(hg, hb, layout=(1,2), size=(800, 400)) # Adjust the size as needed
end
```

```
test_dataset (generic function with 2 methods)
```

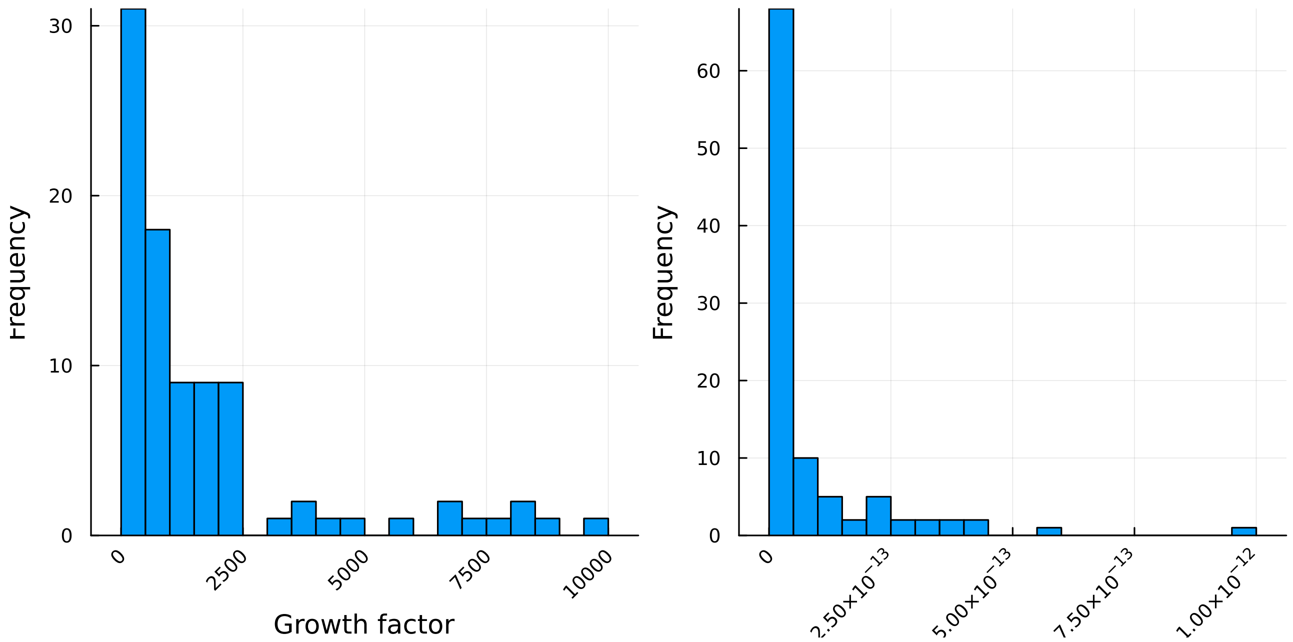
```julia
function test_dataset(dataset_generator, t, plot_quantile=1.)
    Random.seed!(42)
    f = 0
    g = Float64[]
    b = Float64[]
    for i in 1:t
        A = dataset_generator(i)
        try
            L, U, γ = lufact(A)
            β = relative_backward_error(A, L, U)
            push!(g, γ)
            push!(b, β)
        catch e
            if isa(e, DomainError)
                f += 1
            else
                throw(e)
            end
        end
    end
    print_summary(g, b, f, t)
    plot_summary(g, b, plot_quantile)
    #return g, b
end
```

# Random matrices

We generate real matrices of uniformly distributed sizes in the range 2:100 and normal distributed elements.

generate_random (generic function with 1 method)

```
function generate_random(i)
    N = rand(2:100)
    return randn(Float64, N, N)
end
```




```
test_dataset(generate_random, 100, 0.9)
```

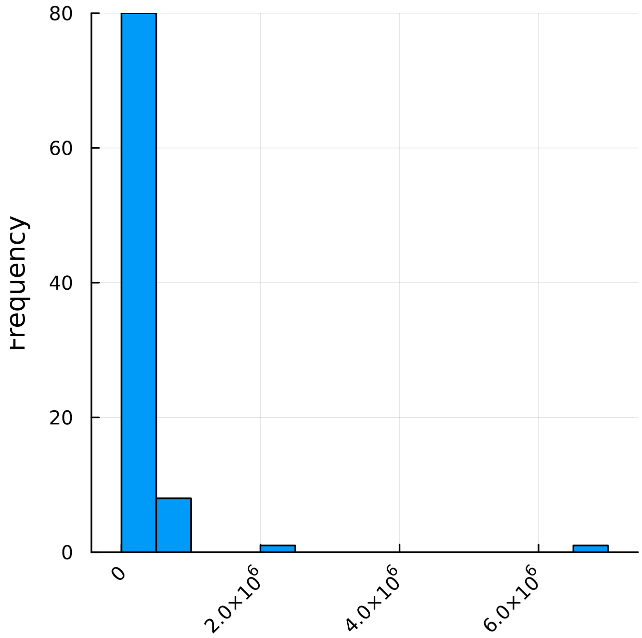
```
Failure rate: 0.0 %
GROWTH FACTOR
Min:    1.0
Max:    47101.04950458209
Mean:   3405.46298922573
Median: 1131.8943097590077
StdDev: 6362.426046577013
RELATIVE BACKWARD ERROR
Min:    0.0
Max:    9.664073393522062e-13
Mean:   8.028283558195911e-14
Median: 2.040374369416543e-14
StdDev: 1.4448853727177071e-13
```

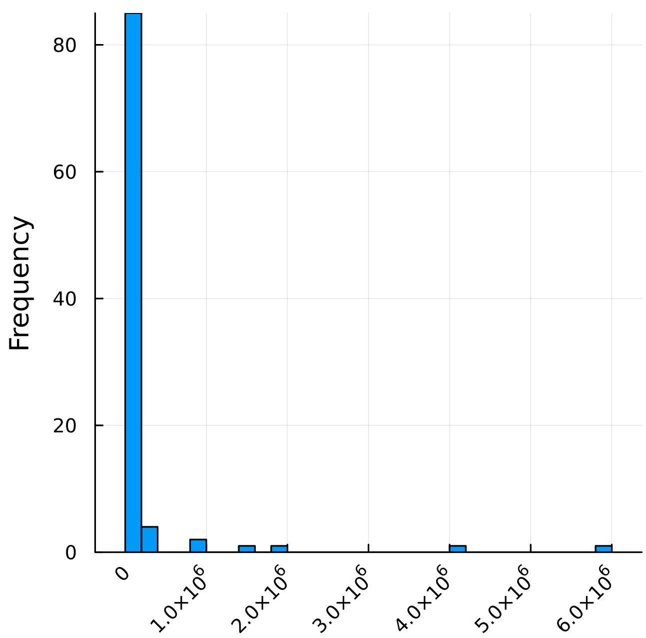
Furthermore, we generate complex matrix with the same size range and elements distribution.

```
md"""
Furthermore, we generate complex matrix with the same size range and elements
distribution.
"""
```

generate_random_complex (generic function with 1 method)

```
function generate_random_complex(i)
    N = rand(2:100)
    return randn(ComplexF64, N, N)
end
```




```
test_dataset(generate_random_complex, 100, 0.9)
```

```
Failure rate: 0.0 %
GROWTH FACTOR
Min:     2.671102767863563
Max:     1.4204629724095856e10
Mean:    1.9728159765561047e8
Median: 19854.886046128588
StdDev: 1.4595643568192155e9
RELATIVE BACKWARD ERROR
Min:     1.249878510419646
Max:     2.2868260696960516e9
Mean:    2.82932720756088e7
Median: 4442.241962951809
StdDev: 2.3071620968102214e8
```

FOR RANDOM IT SEEMS OK. SAY IT BETTER

```
md"""
FOR RANDOM IT SEEMS OK. SAY IT BETTER
"""
```

## Hilbert matrices

Hilber matrices are defined per-element as

$$H_{ij} = \frac{1}{i+j-1}$$

so for example the $4 \times 4$ Hilbert matrix is

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

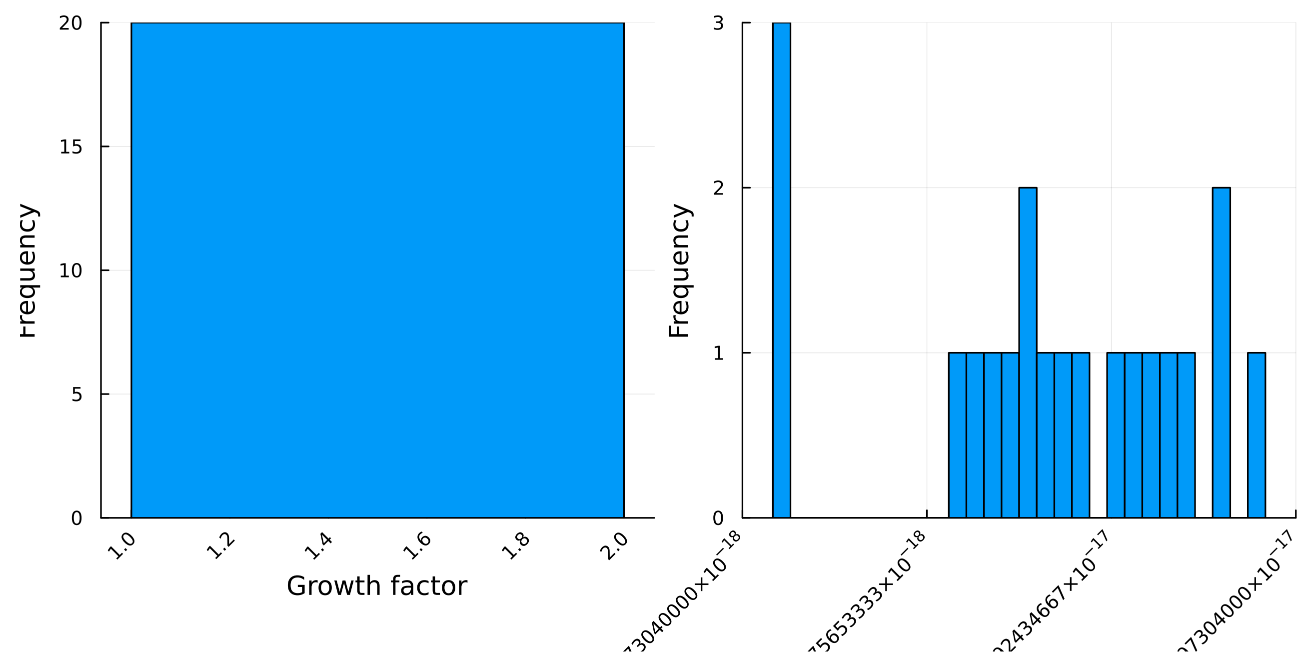
MENTION ILL-CONDITIONED NATURE OF THESE MATRICES

Hence we define the proper generator function.

```
md"""
Hence we define the proper generator function.
"""
```

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

```
function generate_hilbert(N)
    return [1 / (i + j - 1) for i in 1:N, j in 1:N]
end
```




```julia
test_dataset(generate_hilbert, 300)
```

```
Failure rate: 93.33333333333333 %
GROWTH FACTOR
Min:    1.0
Max:    1.0
Mean:   1.0
Median: 1.0
StdDev: 0.0
RELATIVE BACKWARD ERROR
Min:    0.0
Max:    2.7001540943022594e-17
Mean:   1.5533358019968334e-17
Median: 1.5701316941667965e-17
StdDev: 8.271771338970318e-18
```

FANNO CAAA qed

## Diagonally dominant matrices

A square matrix $A$ is said to be diagonally dominant (by rows) if for every row of the matrix, the magnitude of the diagonal entry in a row is greater than or equal to the sum of the magnitudes of all the other (non-diagonal) entries in that row. Mathematically, this can be written as:

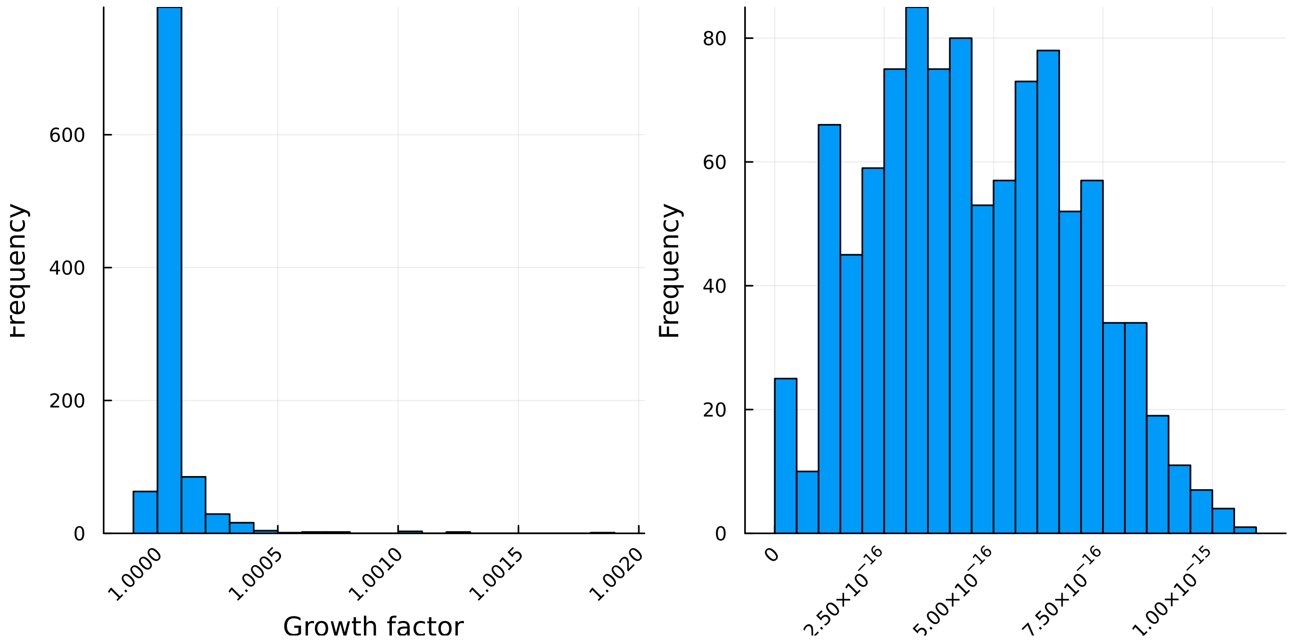
$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \text{for all } i.$$

This condition must hold for each $i$ from $1$ to $n$, where $n$ is the size of the $n \times n$ matrix $A$. If the inequality is strict for all rows, then $A$ is said to be strictly diagonally dominant. The LU factorization without pivoting is known to be backward stable for this irreducible DD matrices.

We built a diagonally dominant generator function by first generating a $N \times N$ matrix with element uniformly distributed in the range 0:1 and then we summed $N + 1$ to the diagonal element.

generate_dd (generic function with 1 method)

```
function generate_dd(i)
    N = rand(2:100)
    A = rand(N, N)
    A += diagm([N+1 for _ in 1:N])
    return A
end
```




```
test_dataset(generate_dd, 1000)
```

```
Failure rate: 0.0 %
GROWTH FACTOR
Min:    0.9999999999999996
Max:    1.0018434432259373
Mean:   1.0000555146830499
Median: 1.000013938106836
StdDev: 0.00012768933341942098
RELATIVE BACKWARD ERROR
Min:    0.0
Max:    1.0565702407697936e-15
Mean:   4.563304322455912e-16
Median: 4.327370074077119e-16
StdDev: 2.2881824703137433e-16
```

stable as qed

**UndefVarError: `plot_correlation` not defined**

1. top-level scope @ *Local: 1*

```
plot_correlation(g3, b3)
```

# SPD matrices

A symmetric matrix $A$ of size $n \times n$ is said to be positive definite if the following two conditions are met:

1. The matrix is symmetric;
2. For any non-zero vector $\mathbf{x}$ in $\mathbb{R}^n$, the quadratic form $\mathbf{x}^T A \mathbf{x}$ is positive, i.e., $\mathbf{x}^T A \mathbf{x} > 0$.

Mathematically, this can be expressed as:

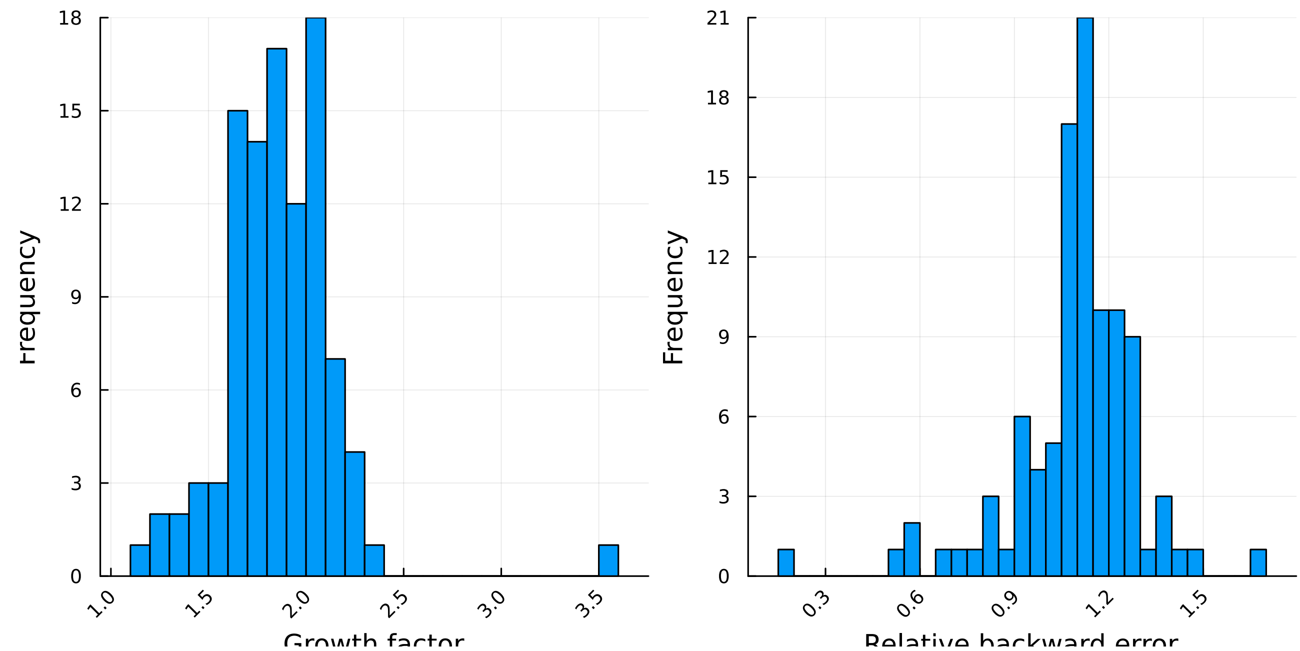
$$\mathbf{x}^T A \mathbf{x} > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}.$$

Additionally, a matrix is positive definite if and only if all its eigenvalues are positive.

```
md"""
A symmetric matrix $A$ of size $n \times n$ is said to be positive definite if the following two conditions are met:

1. The matrix is symmetric;
2. For any non-zero vector $\mathbf{x}$ in $\mathbb{R}^n$, the quadratic form $\mathbf{x}^T A \mathbf{x}$ is positive, i.e., $\mathbf{x}^T A \mathbf{x} > 0$.

Mathematically, this can be expressed as:

$$\mathbf{x}^T A \mathbf{x} > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}.$$

Additionally, a matrix is positive definite if and only if all its eigenvalues are positive.

"""
```

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

```
function generate_spd(i)
    N = rand(2:100)
    A = randn(Complex{Float64}, N, N)
    return A*A'
end
```




```
1  test_dataset(generate_spd, 100)
```

```
Failure rate: 0.0 %
GROWTH FACTOR
Min:    1.1683629837591885
Max:    3.584486788468825
Mean:   1.8637815560595936
Median: 1.866012813410923
StdDev: 0.2892846992223377
RELATIVE BACKWARD ERROR
Min:    0.19547137206541038
Max:    1.6932937729927284
Mean:   1.0965237628900593
Median: 1.1176707045489345
StdDev: 0.2010187457276718
```

commento risultati su stabilita'

an example of performance profiling

```
md"""
an example of performance profiling
"""
```

```
UndefVarError: `A` not defined

 1. var"##core#409"() @ execution.jl:547
 2. var"##sample#410"(::Tuple{}, ::BenchmarkTools.Parameters) @ execution.jl:556
 3. var"#_run#48"(::Bool, ::String, ::Base.Pairs{Symbol, Integer, NTuple{4,
    Symbol}, NamedTuple{(:samples, :evals, :gctrial, :gcsample), Tuple{Int64,
    Int64, Bool, Bool}}}, ::typeof(BenchmarkTools._run),
    ::BenchmarkTools.Benchmark, ::BenchmarkTools.Parameters) @ execution.jl:109
 4. #invokelatest#2 @ essentials.jl:821 [inlined]
 5. invokelatest @ essentials.jl:816 [inlined]
 6. #run_result#45 @ execution.jl:41 [inlined]
 7. run_result @ execution.jl:40 [inlined]
 8. var"#run#49"(::Nothing, ::Float64, ::Float64, ::Base.Pairs{Symbol, Integer,
    NTuple{5, Symbol}, NamedTuple{(:verbose, :samples, :evals, :gctrial,
    :gcsample), Tuple{Bool, Int64, Int64, Bool, Bool}}}, ::typeof(run),
    ::BenchmarkTools.Benchmark, ::BenchmarkTools.Parameters) @ execution.jl:134
 9. run @ execution.jl:126 [inlined]
10. #warmup#54 @ execution.jl:189 [inlined]
11. warmup(::BenchmarkTools.Benchmark) @ execution.jl:188
12. macro expansion @ Local: 432 [inlined]
13. top-level scope @ Local: 12
```

```julia
begin

# Define the range of matrix sizes
n_values = 400:400:4000
# Array to store the benchmark results
timings = []

# Benchmark the lufact() function for different sizes of N
for N in n_values
    A = randn(N, N)
    # Benchmark and get the median time
    bench = @benchmark lufact(A)
    median_time = median(bench).time / 1e9  # Convert to seconds
    push!(timings, median_time)
end

# Plot the timings on a log-log graph
scatter(n_values, timings, label="data", legend=:topleft, xscale=:log10, yscale=:log10, xlabel="n", ylabel="elapsed time (s)")
plot!(n_values, timings[end]*n_values./(n_values[end]).^3, label="O(n^3)", linestyle=:dash, xscale=:log10, yscale=:log10)

end
```

# Problem 2

The Wilkinson matrix is

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 1 \\ -1 & 1 & 0 & \cdots & 1 \\ -1 & -1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & -1 & 1 \end{bmatrix}$$

e qui ci mettiamo due o tre conticini. Sono le otto di sabato, non lo farò ora.

# Task 1

LU decomposition of a Wilkinson Matrix $W_5$

$$W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{bmatrix}$$

# Task 2

Guess of the LU factorization of $W_n$ for a generic $n$

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 \\ -1 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 0 & \cdots & 0 & 2 \\ 0 & 0 & 1 & \cdots & 0 & 2^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 2^{n-1} \end{bmatrix}$$

or

$$L = I_n - \sum_{i=2}^{n} \sum_{j=1}^{i-1} e_i e_j^T$$

$$U = I_n + \sum_{i=1}^{n-1} 2^{i-1} e_i e_n^T$$

# Task 3

introduce function. unite in one?

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

```
function wilkinson_element(i, j, N)
    if i == j || j == N
        return 1.
    elseif i < j
        return 0.
    else
         return -1.
     end
end
```

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

```
function wilkin(N::Integer)
    return [wilkinson_element(i,j,N) for i in 1:N, j in 1:N]
end
```

## Task 4

The vector $\mathbf{b}$ formed by $\mathbf{b} = A\mathbf{e}$ for the Wilkinson matrix $W_n$ where $\mathbf{e}$ is a column vector of ones is given by:

$$\mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ -1 \\ -2 \\ \vdots \\ -(n-3) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ -1 \end{bmatrix}$$

With each entry $b_i$ defined as:

$$b_i = \begin{cases} -(i-3) & \text{for } 1 \leq i < n \\ -(i-2) & \text{for } i = n \end{cases}$$

We thus define a function to store in memory the exact solution to the problem:

```
md"""
### Task 4
The vector $\mathbf{b}$ formed by $\mathbf{b} = A\mathbf{e}$ for the Wilkinson matrix $W_n$ where $\mathbf{e}$ is a column vector of ones is given by:

$$\mathbf{b} = \begin{bmatrix}
2 \\
1 \\
0 \\
-1 \\
-2 \\
\vdots \\
- (n - 3)
\end{bmatrix} + \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
\vdots \\
- 1
\end{bmatrix}$$

With each entry $b_i$ defined as:

$$b_i =
\begin{cases}
- (i - 3) & \text{for } 1 \leq i < n \\
- (i - 2) & \text{for } i = n
\end{cases}$$

We thus define a function to store in memory the exact solution to the problem:
"""
```

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

```julia
function generate_b_vector(n)
    b = [-(i-3) for i in 1:n]
    b[1] = 2
    if n > 1
        b[end] = - (n - 2)
    end
    return b
end
```

```julia
begin
for n in 2:100
    @assert wilkin(n)*[1 for i in 1:n] == generate_b_vector(n)
    #@show wilkin(n)*[1 for i in 1:n]
end

# see which one is faster

#@btime b100 = generate_b_vector(100)
#@btime b100v = generate_b_vector_third(100)
end
```

```
Enter cell code...
```

## Not final , to revise

In task 4, we are asked to perform a numerical experiment with the Wilkinson matrix ( W_n ) and the vector ( e ), which leads to the following steps:

1. Generate ( A = W_n ), the Wilkinson matrix of size ( n \times n ).
2. Let ( e ) be the column vector with all entries equal to 1, then form ( b = Ae ), resulting in a specific pattern as described above.
3. Solve the linear system ( Ax = b ) using the backslash operator, which in Julia (or MATLAB) leverages optimized algorithms for solving linear equations.
4. The computed solution ( x ) is then compared to the exact solution, which should be the vector ( e ).

Given the specific structure of ( W_n ), the backslash operator should ideally find the exact solution without difficulty for smaller values of ( n ). However, as ( n ) becomes large, numerical instability can occur due to the ill-conditioning of the Wilkinson matrix. This can lead to a computed solution ( x ) that deviates from the exact solution ( e ), especially in the presence of round-off errors in floating-point arithmetic.

The success of this numerical experiment heavily depends on the numerical stability of the algorithms used by the backslash operator and the conditioning of the matrix ( W*n ). For ( W*{60} ), we would need to assess the accuracy of the computed solution by comparing it to ( e ) and examining the residual ( r = b - Ax ), which should be close to the zero vector for an accurate solution.

```
AssertionError: x == e

1. top-level scope @ Local: 10 [inlined]
```

```julia
begin
    A = wilkin(60)
    local b = generate_b_vector(60)
    # a 60 dim vector of ones
    e = ones(60)
    x = A \ b
    @show x
    diff = x - e
    @show norm(diff, Inf)
    @assert x == e
end
```

```
x = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
norm(diff, Inf) = 1.0
```

## Task 5 and 6

Repeat the experiment for smaller values of $n$. What is the largest value of $n$ for which $W_n x = b$ can be solved accurately by GEPP when $b = W_n e$? Provide an explanation of the observed behavior.

```
DomainError with relative error introduced at step n = 55:

  1. top-level scope @ Local: 14 [inlined]
```

```julia
# repat the experiment for smaller values of n
# NOTE: for 1 it is false
begin
    for n in 2:60
        Wₙ = wilkin(n)
        b = generate_b_vector(n)
        e = ones(n)
        x = Wₙ \ b

        Δ = x - e
        ϵ_rel = norm(Δ, Inf) / norm(e, Inf)
        if ϵ_rel != 0
            @show ϵ_rel
            throw(DomainError("relative error introduced at step n = $n"))
        end
        @assert x == e
        println("n = $n")
    end
end
```

```
n = 2
n = 3
n = 4
n = 5
n = 6
n = 7
n = 8
n = 9
n = 10
n = 11
n = 12
n = 13
n = 14
n = 15
n = 16
n = 17
n = 18
n = 19
n = 20
n = 21
n = 22
n = 23
n = 24
n = 25
n = 26
n = 27
n = 28
n = 29
n = 30
n = 31
n = 32
n = 33
```

The Wilkinson matrix is specifically designed to be a challenging test case for numerical algorithms due to its structure, which induces a large growth factor ($2^{n-1}$) in the elements of the LU decomposition without pivoting. As the size of the matrix grows, the condition number of the Wilkinson matrix increases exponentially, making it more susceptible to round-off errors.

At $n = 55$ we have a transition: we get a first non-zero entry for the relative error, which becomes 1 under the Infinity norm. Why is that the case? We consider the analytical solution to the problem using the knonw LU factorization for a Wilkinson matrix. Let $L\mathbf{z} = \mathbf{b}$:

We have the following set of equations:

$$\begin{aligned}
z_1 &= b_1, \\
z_2 &= b_2 + z_1 = b_2 + b_1, \\
z_3 &= b_3 + z_2 + z_1 = b_3 + b_2 + 2b_1, \\
z_4 &= b_4 + z_3 + z_2 + z_1 = b_4 + b_3 + 2b_2 + 4b_1, \\
&\vdots
\end{aligned}$$

This pattern continues, and we can generalize it to the equation for any $z_i$:

$$z_i = b_i + b_{i-1} + 2b_{i-2} + \cdots + 2^{i-2}b_1.$$

This can be written more compactly as:

$$z_i = b_i + \sum_{k=1}^{i-1} 2^{k-1} b_{i-k}.$$

We now use the analytical expression for $\mathbf{b}$ to rewrite $z_i$ as:

$$z_i = -(i-3) - \sum_{k=1}^{i-1} 2^{k-1}(i-k-3).$$

excpet for the last $z_n$ where $b_n = -(i-2)$.

We continue from the previous step:

$$-(i-3) - (i-3)\left(\sum_{k=1}^{i-1} 2^{k-1}\right) + \sum_{k=1}^{i-1} k2^{k-1} = \ldots$$

This simplifies to:

$$-(i-3)(2^i - 1) + \sum_{k=1}^{i-1} k2^{k-1} = \ldots$$

And we can conclude with:

$$z_i = 2^{i-1} + 1$$

and

$$z_n = 2^{n-1}$$

DA RIMUOVERE? Finally, we can relate this to the geometric progression sum formula:

$$\sum_{k=0}^{N} 2^k = 2^{N+1} - 1.$$

For example, adding powers of 2 like $1 + 2 + 4 + 8 + 16 + 32$ can be calculated using the sum formula for a geometric series:

$$\sum_{k=0}^{N} 2^k = 2^{N+1} - 1.$$

Now we continue with the backward substitution step, $U\mathbf{x} = \mathbf{z}$, and we know that the following holds for the specific for of U in the Wilkinson matrix:

$$\begin{aligned} x_n &= \frac{z_n}{2^{n-1}} = \frac{2^{n-1}}{2^{n-1}} = 1, \\ &\vdots \\ x_i + 2^{i-1}x_n &= z_i, \rightarrow x_i = z_i - 2^{i-1} = 2^{i-1} + 1 - 2^{i-1} \end{aligned}$$

CONTINUARE QUI

The previous is an exact explanation of the loss of accuracy we observe. However, in order to investigate further this problem, we can recall Theorem 4.29 of the lectures, which states that the relative error for our case is bounded by:

$$\frac{\|\mathbf{x}^* - \tilde{\mathbf{x}}\|}{\|\mathbf{x}^*\|} \leq O(\gamma u)\kappa(W_n)$$

where $\gamma = 2^{n-1}$ for the Wilkinson matrix. We can thus compute the upper bound of ϵ_rel in the vicinity of $n = 55$ to convince ourselves that the problem is unstable in this regime:

```julia
begin
    for n in 45:60
    Wₙ = wilkin(n)
    κ = cond(Wₙ, Inf)
    γ = 2^(n-1)
    ϵ_bound = γ * eps()/2 * κ
    println("At n = $n ϵ_bound ≈ $ϵ_bound")
    end
end
```

```
At n = 45 ϵ_bound ≈ 0.087890625
At n = 46 ϵ_bound ≈ 0.1796875
At n = 47 ϵ_bound ≈ 0.3671875
At n = 48 ϵ_bound ≈ 0.75
At n = 49 ϵ_bound ≈ 1.53125
At n = 50 ϵ_bound ≈ 3.125
At n = 51 ϵ_bound ≈ 6.375
At n = 52 ϵ_bound ≈ 13.0
At n = 53 ϵ_bound ≈ 26.5
At n = 54 ϵ_bound ≈ 54.0
At n = 55 ϵ_bound ≈ 110.0
At n = 56 ϵ_bound ≈ 224.0
At n = 57 ϵ_bound ≈ 456.0
At n = 58 ϵ_bound ≈ 928.0
At n = 59 ϵ_bound ≈ 1888.0
At n = 60 ϵ_bound ≈ 3840.0
```

We see that the relative error bound, ϵ_bound, is of order 1 at about $n = 48$. After that, numerical optimizations make so that the solution can still be computed accurately, but we do not have any guarantee from theory.

# Problem 3

## Task 1

### Point a)

We want to prove that the matrix $\tilde{A} = A + \mathbf{u}\mathbf{v}^T$ is nonsingular if and only if $\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$.

To start, let's remember that a matrix is nonsingular (or invertible) if it has a nonzero determinant.

First we introduce the matrix determinant lemma which states that for any invertible $n \times n$ matrix $A$ and column vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, the determinant of $A + \mathbf{u}\mathbf{v}^T$ is given by:

$$\det(A + \mathbf{u}\mathbf{v}^T) = \det(A) \cdot (1 + \mathbf{v}^T A^{-1}\mathbf{u})$$

**Proof of the Matrix Determinant Lemma**

We start from the special case $A = I$. Let's define the bordered matrix

$$\begin{pmatrix} I + \mathbf{u}\mathbf{v}^T & \mathbf{u} \\ 0 & 1 \end{pmatrix}$$

the following identity holds:

$$\begin{pmatrix} I & 0 \\ \mathbf{v}^T & 1 \end{pmatrix} \begin{pmatrix} I + \mathbf{u}\mathbf{v}^T & \mathbf{u} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} I & 0 \\ -\mathbf{v}^T & 1 \end{pmatrix} = \begin{pmatrix} I & \mathbf{u} \\ 0 & 1 + \mathbf{v}^T\mathbf{u} \end{pmatrix}$$

The first and third matrix on the left hand side of the equation are unit lower triangular matrices, thus their determinants are 1. Since the determinant of a block triangular matrix is the product of the determinants of the diagonal blocks, we have:

$$\det(I + \mathbf{u}\mathbf{v}^T) = (1 + \mathbf{v}^T\mathbf{u}).$$

It is simple the to derive the general case:

$$\begin{aligned} \det(A + \mathbf{u}\mathbf{v}^T) &= \det(A)\det(I + (A^{-1}\mathbf{u})\mathbf{v}^T) \\ &= \det(A)(1 + \mathbf{v}^T(A^{-1}\mathbf{u})). \end{aligned}$$

And this ends the proof of the Matrix Determinant Lemma.

For $\tilde{A}$ to be nonsingular, $\det(\tilde{A})$ must be nonzero. Based on the matrix determinant lemma, this will be the case if and only if:

$$1 + \mathbf{v}^T A^{-1}\mathbf{u} \neq 0$$

Then, for $\tilde{A}$ to be invertible,

$$\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$$

And this concludes the proof.

## Point b)

We want to derive the Sherman Morrison formula:

$$\tilde{A}^{-1} = A^{-1} - \alpha A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}, \qquad \alpha = \frac{1}{1+\mathbf{v}^T A^{-1}\mathbf{u}}$$

To do so, let's assume that $\tilde{A}^{-1}$ is of the form $A^{-1}+B$. Then

$$\begin{aligned}\tilde{A}\tilde{A}^{-1} = I &= (A+\mathbf{u}\mathbf{v}^T)(A^{-1}+B) \\ &= I + \mathbf{u}\mathbf{v}^T A^{-1} + (A+\mathbf{u}\mathbf{v}^T)B\end{aligned}$$

and hence

$$\begin{aligned}B &= -(A+\mathbf{u}\mathbf{v}^T)^{-1}\mathbf{u}\mathbf{v}^T A^{-1} \\ &= -(A^{-1}+B)\mathbf{u}\mathbf{v}^T A^{-1}.\end{aligned}$$

After some manipulation, we find that

$$(1+\mathbf{v}^T A^{-1}\mathbf{u})B = A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}$$

Therefore, we can isolate $B$ on one side of the equation:

$$B = -\alpha A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}$$

With $\alpha$ defined as above: the Sherman Morrison formula is found.

## Point c)

Since we have the $LU$ factorization of $A$, let's define $\mathbf{x}$ as the solution of the problem $A\mathbf{x} = \tilde{\mathbf{b}}$ which can be found by backward-forward susbstitution with a complexity $n^2$.

Now, our goal is to find the solution $\tilde{\mathbf{x}}$ to the problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. Formally, we have $\tilde{\mathbf{x}} = \tilde{A}^{-1}\tilde{\mathbf{b}}$ but by using the Sherman Morrison formula we can write

$$\begin{aligned}\tilde{\mathbf{x}} &= (A^{-1} - \alpha A^{-1}\mathbf{u}\mathbf{v}^T A^{-1})\tilde{\mathbf{b}} \\ &= A^{-1}\tilde{\mathbf{b}} - \alpha A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}\tilde{\mathbf{b}} \\ &= \mathbf{x} - \alpha A^{-1}\mathbf{u}\mathbf{v}^T\mathbf{x}\end{aligned}$$

$$A^{-1}\mathbf{u} \equiv \mathbf{y}$$

can be found using the $LU$ decomposition with backward forward substituion as well (quadratic complexity). The remaining operations are vector dot products or vector sums, which are of linear complexity.

To conclude, the following algorithm then finds $\tilde{\mathbf{x}}$ with $\mathcal{O}(n^2)$ flops:

1. Find $\mathbf{x}$ by forward backward substitution;
2. Find $\mathbf{y}$ by forward backward substitution;
3. Compute $\alpha = 1/(1 + \mathbf{v}^T\mathbf{y})$;
4. Compute $\tilde{\mathbf{x}} = \mathbf{x} - \alpha\mathbf{y}(\mathbf{v}^T\mathbf{x})$.

# Task 2

The bordered system

QUA CI PENSA GPTGPTGPTGPT

corresponds to the system

ANCHE QUA BRUMBRUMBRUM

Then, by substitution of $z$, we can recounduce to the previous case, that has quadratic complexity.