Numerical Analysis and Optimization Homework Project 1

This assignment is done using the Julia programming language. To easy the task, we load Julia's linear algebra standard library:

```
using LinearAlgebra, Plots, Statistics
```

Problem 1

Here we define the function `lufact` that takes as input a squatre matrix and compute the non-pivoted LU factorization and its grow factor:

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

```
function lufact(A::AbstractMatrix{T}) where T <: Real
    # Sanity checks
    if size(A, 1) != size(A, 2)
        throw(ArgumentError("The matrix is not squared."))
    end
    n = size(A, 1)
    a = copy(A)
    γ = zero(T)
    for i in 1:n-1
        # Sanity checks
        if abs(a[i,i]) <= eps(T)
            throw(DomainError("Trying to divide by a number smaller than the
            machine precision."))
        end
        # For the U L computation, we exploit Julia's bootstrap syntax.
        # L computation
        a[i+1:end,i] .= a[i+1:end,i] ./ a[i,i]
        if any(isnan, a[i+1:end,i])
            throw(DomainError("Something happened but to know we have to compute
            grow factor."))
        end
        # U computation
        a[i+1:end,i+1:end] .-= a[i+1:end,i] .* a[i+1:end,i+1:end]
        # lacks the gamma computation
    end
    # Note that UpperTriangular and UnitLowerTriangular are not
    # copies of a, but views instead.
    return UnitLowerTriangular(a), UpperTriangular(a), γ
end
```

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

```
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
```

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

```
function plot_summary(g, b)
    hg = histogram(g, xlabel="Growth factor", ylabel="Frequency")
    hb = histogram(b, xlabel="Relative backward error", ylabel="Frequency")
    plot(hg, hb, layout=(1,2))
end
```

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

```
function print_summary(g, b, f)
    println("Failure rate: $f")
    println("GROWTH FACTOR")
    println("Min:    $(minimum(g))")
    println("Max:    $(maximum(g))")
    println("Mean:   $(mean(g))")
    println("StdDev: $(std(g))")
    println("RELATIVE BACKWARD ERROR")
    println("Min:    $(minimum(b))")
    println("Max:    $(maximum(b))")
    println("Mean:   $(mean(b))")
    println("StdDev: $(std(b))")
end
```

Random matrices

```
begin
    f = 0
    g = Float64[]
    b = Float64[]
    for _ in 1:100
        N = rand(2:100)
        A = randn(N, N)
        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)
end
```

```
Failure rate: 83
GROWTH FACTOR
Min:    0.0
Max:    0.0
Mean:   0.0
StdDev: 0.0
RELATIVE BACKWARD ERROR
Min:    NaN
Max:    NaN
Mean:   NaN
StdDev: NaN
```

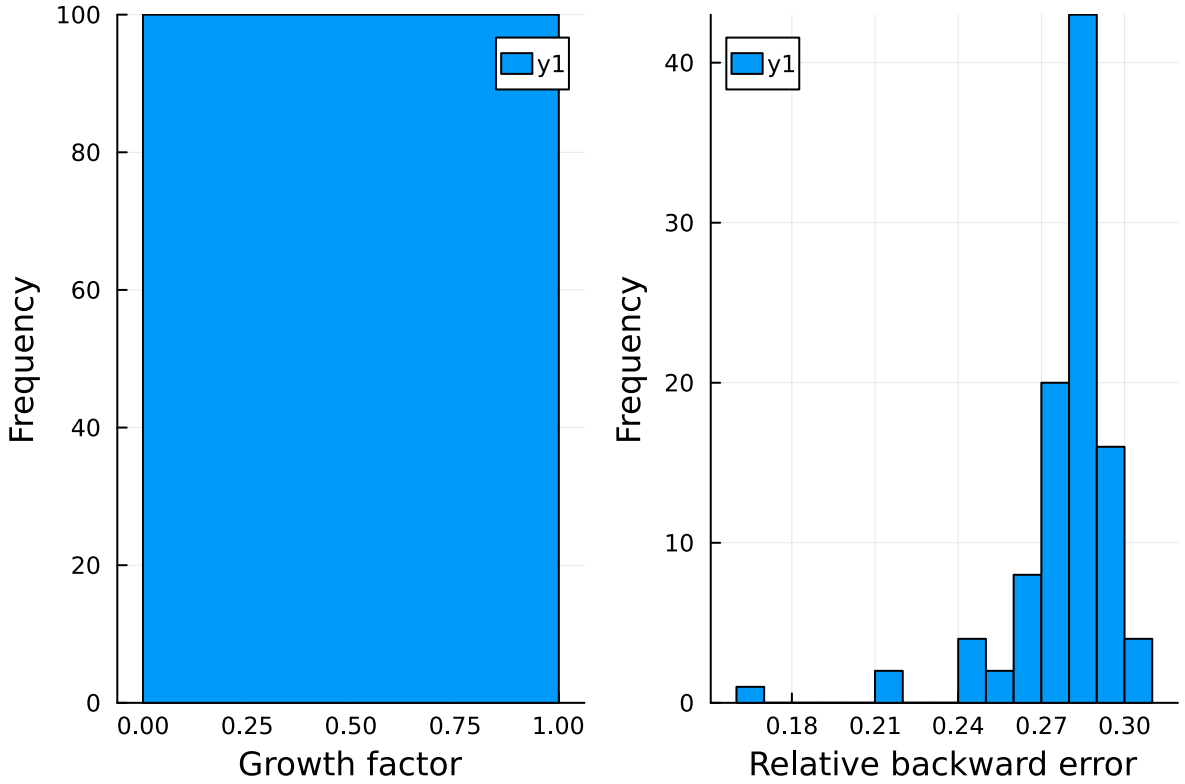
Hilbert matrices

Per/con Francesco ;)

Diagonally dominant matrices

```
begin
    f3 = 0
    g3 = Float64[]
    b3 = Float64[]
    for _ in 1:100
        N = rand(2:100)
        A = rand(N, N)
        A += diagm([N+1 for _ in 1:N])
        try
            L, U, γ = lufact(A)
            β = relative_backward_error(A, L, U)
            push!(g3, γ)
            push!(b3, β)
        catch e
            if isa(e, DomainError)
                f3 += 1
            else
                throw(e)
            end
        end
    end
    print_summary(g3, b3, f3)
end
```

```
Failure rate: 0
GROWTH FACTOR
Min:    0.0
Max:    0.0
Mean:   0.0
StdDev: 0.0
RELATIVE BACKWARD ERROR
Min:    0.16611829155959806
Max:    0.30410374071806945
Mean:   0.27857773932733654
StdDev: 0.019433086674748343
```




```
1 plot_summary(g3, b3)
```

FORSE SAREBBE GANZO FARE UN PLOT DI CORRELAZIONE??

SVD matrices

```
begin
    f4 = 0
    g4 = Float64[]
    b4 = Float64[]
    for _ in 1:100
        N = rand(2:100)
        A = randn(N, N)
        A = A'*A
        try
            L, U, γ = lufact(A)
            β = relative_backward_error(A, L, U)
            push!(g4, γ)
            push!(b4, β)
        catch e
            if isa(e, DomainError)
                f4 += 1
            else
                throw(e)
            end
        end
    end
    print_summary(g4, b4, f4)
end
```

```
Failure rate: 68
GROWTH FACTOR
Min:    0.0
Max:    0.0
Mean:   0.0
StdDev: 0.0
RELATIVE BACKWARD ERROR
Min:    NaN
Max:    NaN
Mean:   NaN
StdDev: NaN
```

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.

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
```