Table of Contents

Algorithm 2.7 Cholesky Factorization
Banded matrices
Sparse matrices and linear operators
Rank 1 update: Sherman-Morrison formula

Assignments

present

About assignments

1. You should submit your homework through Github pull request. Xuanzhao Gao will comment on your PR, then you should resolve the issues to get your PR merged. You should submit different PRs for different assignments. Open repo
2. We will grade based on your merged PRs. You will not be failed if your submition is complete.
3. You may check the answers of other students, but you must credit him in your PR description, e.g.

```
### Reference:
* PR #3
* PR #4
```

System of Linear Equations

$$Solving: \quad \mathbf{Ax} = \mathbf{b}$$

Quiz: In a cage, chickens and rabbits add up to 35 heads and 94 feet. Please count the number of chickens and rabbits.

Table of contents

- Gaussian elimination algorithm
- Pivoting technique
- Sensitivity analysis and condition number
- Getting matrix inverse with Gauss-Jordan elimination
- Solving linear equations for special matrices (optional)

Schetch of solving the linear equation

$$
\begin{aligned}
&A = LU \\
&x = A^{-1}b = U^{-1}L^{-1}b
\end{aligned}
$$

Solving tridiagonal linear equation

$$Lx = b$$

$$L = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix}$$

Algorithm 2.1: Forward-Substitution for Lower Triangular System

$$x_1 = b_1/l_{11}, \quad x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j \right)/l_{ii}, \;\; i = 2, \ldots, n$$

```
back_substitution! (generic function with 1 method)
```

```julia
function back_substitution!(l::AbstractMatrix, b::AbstractVector)
    n = length(b)
    @assert size(l) == (n, n) "size mismatch"
    x = zero(b)
    # loop over columns
    for j = 1:n
        # stop if matrix is singular
        if iszero(l[j, j])
            error("The lower triangular matrix is singular!")
        end
        # compute solution component
        x[j] = b[j] / l[j, j]
        for i = j+1:n
            # update right hand side
            b[i] = b[i] - l[i, j] * x[j]
        end
    end
    return x
end
```

```
l = 4×4 Matrix{Float64}:
  0.547284   0.0        0.0        0.0
 -1.068      0.759989   0.0        0.0
 -0.197166  -0.345128  -0.81142    0.0
  0.735737  -0.971137  -0.300388  -0.0411604
```

```julia
l = tril(randn(4, 4))
```

```
b = [-0.930708, -0.623271, -0.471425, -0.917524]
```

```julia
b = randn(4)
```

```
[-1.7006, -3.20992, 2.35952, 50.4085]
```

```julia
back_substitution!(l, copy(b))
```

The Julia's version

```julia
using LinearAlgebra
```

```
[-1.7006, -3.20992, 2.35952, 50.4085]
```

```julia
LowerTriangular(l) \ b
```

LU Factorization

$$A = LU$$

The Gaussian elimination process

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

The elementary elimination matrix

$$M_1 A = \begin{pmatrix} a_{11} & a_{12} & \dots \\ 0 & a'_{22} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$M_k = \begin{pmatrix} 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & -m_{k+1} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -m_n & 0 & \dots & 1 \end{pmatrix}$$

where $m_i = a_i / a_k$.

Then

$$L = M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1}$$

Properties of elimination matrices

$$M_k^{-1} = 2I - M_k$$

$$M_k M_{k'>k} = M_k + M_{k'} - I$$

Coding: Properties of elimination matrices

```
A3 = 3×3 Matrix{Int64}:
      1  2  2
      4  4  2
      4  6  4
```

```
A3 = [1 2 2; 4 4 2; 4 6 4]
```

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

```
function elementary_elimination_matrix(A::AbstractMatrix{T}, k::Int) where T
    n = size(A, 1)
    @assert size(A, 2) == n
    # create Elementary Elimination Matrices
    M = Matrix{Float64}(I, n, n)
    for i=k+1:n
        M[i, k] =  -A[i, k] ./ A[k, k]
    end
    return M
end
```

The elimination

```
3×3 Matrix{Float64}:
  1.0  0.0  0.0
 -4.0  1.0  0.0
 -4.0  0.0  1.0
```

```
elementary_elimination_matrix(A3, 1)
```

```
3×3 Matrix{Float64}:
 1.0   2.0   2.0
 0.0  -4.0  -6.0
 0.0  -2.0  -4.0
```

```
elementary_elimination_matrix(A3, 1) * A3
```

The inverse

```
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 4.0  1.0  0.0
 4.0  0.0  1.0
```

```
inv(elementary_elimination_matrix(A3, 1))
```

The multiplication

```
3×3 Matrix{Float64}:
 1.0   0.0  0.0
 0.0   1.0  0.0
 0.0  -1.5  1.0
```

```
elementary_elimination_matrix(A3, 2)
```

```
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 4.0  1.0  0.0
 4.0  1.5  1.0
```

```julia
inv(elementary_elimination_matrix(A3, 1)) * inv(elementary_elimination_matrix(A3, 2))
```

Algorithm 2.3: LU Factorization by Gaussian Elimination

```
lufact_naive! (generic function with 1 method)
```

```julia
function lufact_naive!(A::AbstractMatrix{T}) where T
    n = size(A, 1)
    @assert size(A, 2) == n
    M = Matrix{T}(I, n, n)
    for k=1:n-1
        m = elementary_elimination_matrix(A, k)
        M = M * inv(m)
        A .= m * A
    end
    return M, A
end
```

```
(3×3 Matrix{Float64}:, 3×3 Matrix{Int64}:)
  1.0  0.0  0.0          1   2   2
```

```julia
lufact_naive!(copy(A3))
```

```julia
using Test
```

```
Test Passed
```

```julia
let
    A = [1 2 2; 4 4 2; 4 6 4]
    L, U = lufact_naive!(copy(A))
    @test L * U ≈ A
end
```

Better implementation

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

```julia
function lufact!(a::AbstractMatrix)
    n = size(a, 1)
    @assert size(a, 2) == n "size mismatch"
    m = zero(a)
    m[1:n+1:end] .+= 1
    # loop over columns
    for k=1:n-1
        # stop if pivot is zero
        if iszero(a[k, k])
            error("Gaussian elimination fails!")
        end
        # compute multipliers for current column
        for i=k+1:n
            m[i, k] = a[i, k] / a[k, k]
        end
        # apply transformation to remaining sub-matrix
        for j=k+1:n
            for i=k+1:n
                a[i,j] -= m[i,k] * a[k, j]
            end
        end
    end
    return m, triu!(a)
end
```

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

```julia
lufact(a::AbstractMatrix) = lufact!(copy(a))
```

```
DefaultTestSet("LU", [], 3, false, false, true, 1.677652817066939e9, 1.677652817066971e9)
```

```julia
@testset "LU" begin
    a = randn(4, 4)
    L, U = lufact(a)
    @test istril(L)
    @test istriu(U)
    @test L * U ≈ a
end
```

```
Test Summary: | Pass  Total  Time
LU            |    3      3  0.0s
```

```
A4 = 4×4 Matrix{Float64}:
 -1.13722    0.32243    1.17589   0.309568
 -0.499461   0.502538  -0.212333  0.325516
  0.291973   0.860287  -1.58332   0.191323
  0.79411   -0.58919    0.293682  0.449737
```

```julia
A4 = randn(4, 4)
```

```
(4×4 Matrix{Float64}:                    , 4×4 Matrix{Float64}:                    )
 1.0        0.0       0.0       0.0    -1.13722  0.32243    1.17589    0.309568
```

```julia
lufact(A4)
```

The Julia's version

```
julia_lures = LU{Float64, Matrix{Float64}, Vector{Int64}}
              L factor:
              4×4 Matrix{Float64}:
                1.0        0.0       0.0       0.0
                0.439196   1.0       0.0       0.0
               -0.256744   2.6129    1.0       0.0
               -0.698293  -1.00862   0.609723  1.0
              U factor:
              4×4 Matrix{Float64}:
               -1.13722  0.32243    1.17589    0.309568
                0.0      0.360928  -0.728778   0.189556
                0.0      0.0        0.622801  -0.224488
                0.0      0.0        0.0        0.993971
```

```
julia_lures = lu(A4, NoPivot())  # the version we implemented above has no pivot
```

```
4×4 Matrix{Float64}:
 -1.13722  0.32243    1.17589    0.309568
  0.0      0.360928  -0.728778   0.189556
  0.0      0.0        0.622801  -0.224488
  0.0      0.0        0.0        0.993971
```

```
julia_lures.U
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
```

```
typeof(julia_lures)
```

```
(:factors, :ipiv, :info)
```

```
fieldnames(julia_lures |> typeof)
```

Complexity Analysis

$$O(n^3)$$

Issue: how to handle small diagonal entries?

-2.0

```
@bind ϵ Slider(-20:0.01:0.0; default=-2, show_value=true)
```

```
small_diagonal_matrix = 2×2 Matrix{Float64}:
                         1.0e-20  1.0
                         1.0      1.0
```

```
small_diagonal_matrix = [10^(ϵ) 1; 1 1]
```

```
lures =  (2×2 Matrix{Float64}:, 2×2 Matrix{Float64}:)
           1.0     0.0         1.0e-20   1.0
```

```
lures = lufact(small_diagonal_matrix)
```

A better approach

```
(2×2 Matrix{Float64}:, 2×2 Matrix{Float64}:)
  1.0      0.0          1.0  1.0
```

```
lufact(small_diagonal_matrix[end:-1:1, :])
```

Pivoting technique

$$PA = LU$$

```
2×2 Matrix{Float64}:
 1.0      1.0
 1.0e-20  1.0
```

```
[10^(ϵ) 1; 1 1][end:-1:1, :]
```

Gaussian Elimination process with Partial Pivoting

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\ldots M_2 P_2 M_1 P_1 A = U$$

NOTE: P_{k+1} and M_k commute

Algorithm 2.4 LU Factoriazation by Gaussian Elimination with Partial Pivoting

```
lufact_pivot! (generic function with 1 method)
```

```julia
function lufact_pivot!(a::AbstractMatrix)
    n = size(a, 1)
    @assert size(a, 2) == n "size mismatch"
    m = zero(a)
    P = collect(1:n)
    # loop over columns
    @inbounds for k=1:n-1
        # search for pivot in current column
        val, p = findmax(x->abs(a[x, k]), k:n)
        p += k-1
        # find index p such that |a_{pk}| ≥ |a_{ik}| for k ≤ i ≤ n
        if p != k
            # swap rows k and p of matrix A
            for col = 1:n
                a[k, col], a[p, col] = a[p, col], a[k, col]
            end
            # swap rows k and p of matrix M
            for col = 1:k-1
                m[k, col], m[p, col] = m[p, col], m[k, col]
            end
            P[k], P[p] = P[p], P[k]
        end
        if iszero(a[k, k])
            # skip current column if it's already zero
            continue
        end
        # compute multipliers for current column
        m[k, k] = 1
        for i=k+1:n
            m[i, k] = a[i, k] / a[k, k]
        end
        # apply transformation to remaining sub-matrix
        for j=k+1:n
            akj = a[k, j]
            for i=k+1:n
                a[i,j] -= m[i,k] * akj
            end
        end
    end
    m[n, n] = 1
    return m, triu!(a), P
end
```

```
Test Passed
```

```julia
let
    n = 5
    A = randn(n, n)
    L, U, P = lufact_pivot!(copy(A))
    pmat = zeros(Int, n, n)
    setindex!.(Ref(pmat), 1, 1:n, P)
    @test L ≈ lu(A).L
    @test U ≈ lu(A).U
    @test pmat * A ≈ L * U
end
```

```julia
using BenchmarkTools
```

☐

```julia
@bind benchmark_lu CheckBox()
```

```julia
if benchmark_lu let
    n = 200
    A = randn(n, n)
    @benchmark lufact_pivot!($A)
end end
```

```julia
if benchmark_lu let
    n = 200
    A = randn(n, n)
    @benchmark lu($A)
end end
```

Complete pivoting

$$PAQ = LU$$

Sensitivity Analysis

Issue: An Ill Conditioned Matrix

$$A = \begin{pmatrix} 0.913 & 0.659 \\ 0.457 & 0.330 \end{pmatrix}$$

```
ill_conditioned_matrix = 2×2 Matrix{Float64}:
                          0.913  0.659
                          0.457  0.33
```

```
ill_conditioned_matrix = [0.913 0.659; 0.457 0.330]
```

```
lures2 =   (2×2 Matrix{Float64}:, 2×2 Matrix{Float64}:)
            1.0       0.0         0.913  0.659
```

```
lures2 = lufact(ill_conditioned_matrix)
```

```
(2×2 Matrix{Float64}:, 2×2 Matrix{Float64}:)
 1.0       0.0         0.913  0.659
```

```
lures2
```

```
12485.031415973668
```

```
cond(ill_conditioned_matrix)
```

Forward Error and Backward Error

Forward error:

$$\mathrm{dist}(f(x), \hat{f}(x))$$

Backward error

$$\mathrm{dist}(x, f^{-1}(\hat{f}(x)))$$

Absolute Error and Relative Error

Absolute error: $\|x - \hat{x}\|$

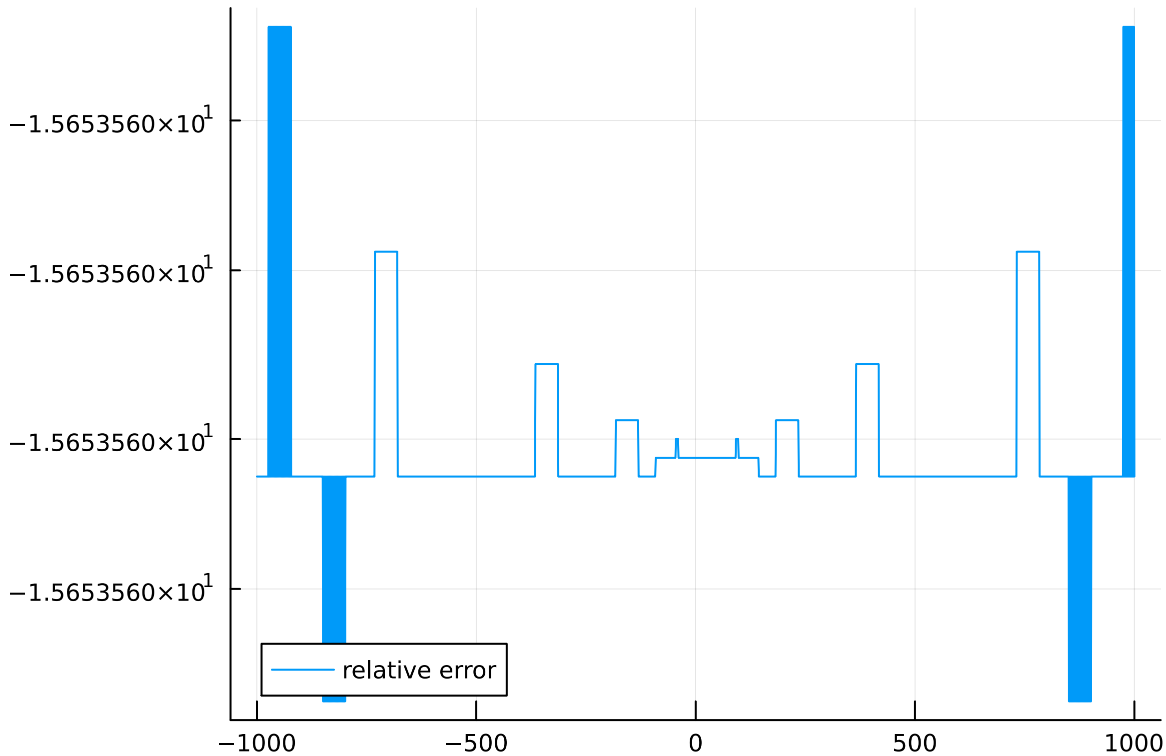
Relative error: $\frac{\|x-\hat{x}\|}{\|x\|}$

where $\|\cdot\|$ is a measure of size.

Coding: Floating point numbers have "constant" relative error

```
2.220446049250313e-16
```

```
eps(Float64)
```




```
let
    n = 1000
    reltol = zeros(2n+1)
    for i=-n:n
        f = 2.0^i
        reltol[i+n+1] = log10(eps(f)) - log10(f)
    end
    plot(-n:n, reltol; label="relative error")
end
```

```
GKS: Possible loss of precision in routine SET_WINDOW
```

2.220446049250313e-16

```
eps(1.0)/1.0
```

2.220446049250313e-16

```
eps(2.0)/2.0
```

1.570092458683775e-16

```
eps(sqrt(2))/sqrt(2)
```

(Relative) Condition Number

$$\lim_{\varepsilon \to 0^+} \sup_{\|\delta x\| \leq \varepsilon} \frac{\|\delta f(x)\|/\|f(x)\|}{\|\delta x\|/\|x\|}$$

Quantify Error of a Scalar Operator

$$y = \exp(x)$$

Why we should avoid using floating point numbers being too big/small?

$$a + b$$

Measuring the size of a vector: Norms

$$\|v\|_p = \left(\sum_i |v_i|^p\right)^{1/p}$$

Measure the size of a matrix

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

Coding: Vector and Matrix Norms

```
5.0
```

```julia
norm([3, 4], 2)
```

```
7.0
```

```
norm([3, 4], 1)
```

```
4.0
```

```
norm([3, 4], Inf)
```

```
2.0
```

```
# l0 norm is not a true norm
norm([3, 4], 0)
```

```
1.0
```

```
norm([3, 0], 0)
```

```
mat = 2×2 Matrix{Float64}:
       -0.284666   -1.24104
       -0.0321259   2.3657
```

```
mat = randn(2, 2)
```

```
3.6067408469604665
```

```
opnorm(mat, 1)
```

```
2.3978261092617075
```

```
opnorm(mat, Inf)
```

```
2.673499761890822
```

```
opnorm(mat, 2)
```

```
ArgumentError: invalid p-norm p=0. Valid: 1, 2, Inf

  1. opnorm(::Matrix{Float64}, ::Int64) @ generic.jl:740
  2. top-level scope @ Local: 1  [inlined]
```

```
opnorm(mat, 0)
```

```
10.020429724445629
```

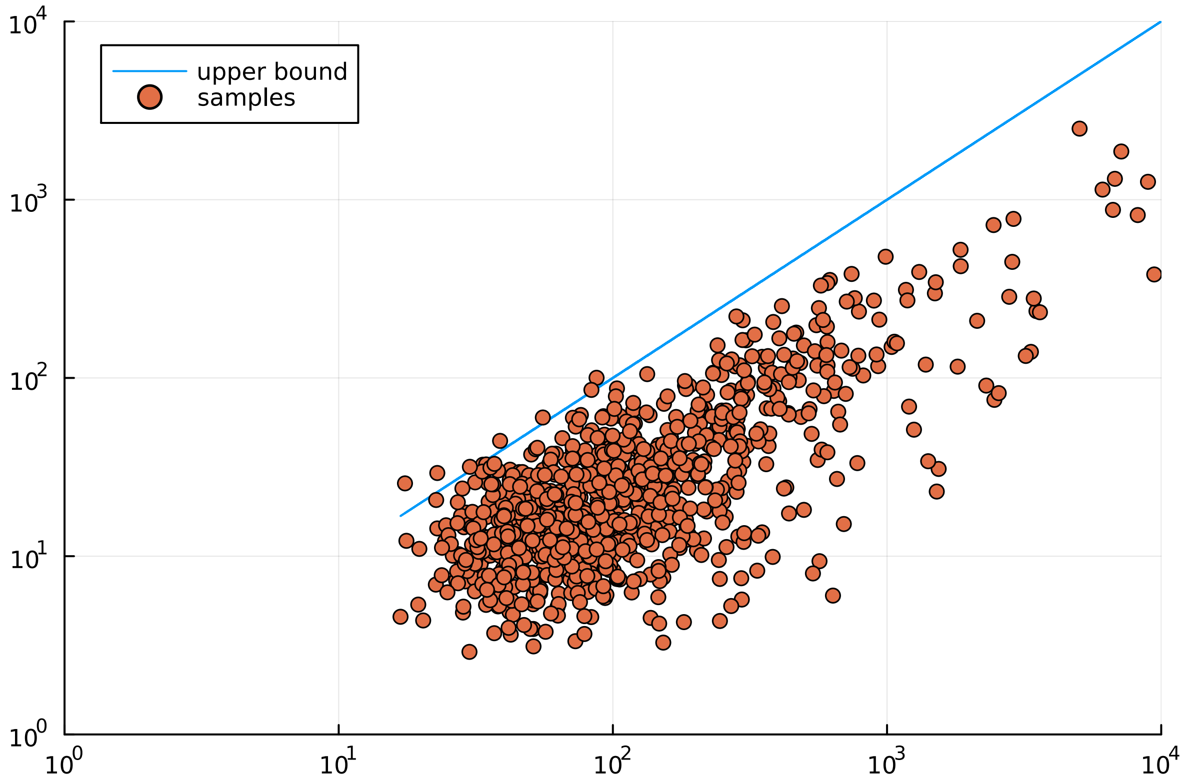
```
cond(mat)
```

Condition Number of a Linear Operator

$$\mathrm{cond}(A) = \|A\|\|A^{-1}\|$$

Coding: Numeric experiment on condition number

```julia
using Plots
```




```julia
let
    n = 1000
    p = 2
    errors = zeros(n)
    conds = zeros(n)
    for k = 1:n
        A = rand(10, 10)
        b = rand(10)
        dx = A \ b
        sx = Float32.(A) \ Float32.(b)
        errors[k] = (norm(sx - dx, p)/norm(dx, p)) / (norm(b-Float32.(b), p)/norm(b,
        p))
        conds[k] = cond(A, p)
    end
    plt = plot(conds, conds; label="upper bound", xlim=(1, 10000), ylim=(1, 10000),
    xscale=:log10, yscale=:log10)
    scatter!(plt, conds, errors; label="samples")
end
```

Computing Matrix Inverse

Gauss Jordan Elimination Matrix

$$N_k = \begin{pmatrix} 1 & \dots & 0 & -m_1 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & -m_{k-1} & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & -m_{k+1} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -m_n & 0 & \dots & 1 \end{pmatrix}$$

where $m_i = a_i/a_k$.

Computing the inverse

$$SN_nN_{n-1}\dots N_1A = I$$

Then

$$A^{-1} = SN_nN_{n-1}\dots N_1$$

Special Matrices

Positive definite symmetric matrix

- (Real) Symmetric: $A = A^T$,
- Positive definite: $x^TAx > 0$ for all $x \neq 0$.

Cholesky decomposition

$$A = LL^T$$

Algorithm 2.7 Cholesky Factorization

```
chol! (generic function with 1 method)
```

```julia
function chol!(a::AbstractMatrix)
    n = size(a, 1)
    @assert size(a, 2) == n
    for k=1:n
        a[k, k] = sqrt(a[k, k])
        for i=k+1:n
            a[i, k] = a[i, k] / a[k, k]
        end
        for j=k+1:n
            for i=k+1:n
                a[i,j] = a[i,j] - a[i, k] * a[j, k]
            end
        end
    end
    return a
end
```

```
10×10 Matrix{Float64}:
 -5.55112e-17  -2.77556e-17  -2.77556e-16  …   1.38778e-17  -2.77556e-17
  0.0          -1.11022e-16   2.77556e-17      5.89806e-17  -1.11022e-16
  0.0           0.0           0.0             -1.11022e-16  -6.93889e-17
  0.0           0.0           2.77556e-17      5.55112e-17  -6.93889e-18
  0.0           0.0           0.0             -1.63389e-16   4.16334e-17
  0.0           1.6263e-19    0.0          …  -2.08167e-17   1.71738e-16
  0.0          -2.77556e-17   1.38778e-17     -5.55112e-17   1.249e-16
  0.0           0.0          -6.93889e-18      0.0          -8.32667e-17
  0.0           0.0           2.77556e-17      5.55112e-17   2.77556e-17
 -1.38778e-17   0.0           0.0              0.0           0.0
```

```julia
let
    n = 10
    Q, R = qr(randn(10, 10))
    a = Q * Diagonal(rand(10)) * Q'
    L = chol!(copy(a))
    tril(L) * tril(L)' - a
    # cholesky(a) in Julia
end
```

Banded matrices

Sparse matrices and linear operators

Accessing every element is not allowed, but matrix-vector multiplication is defined.

You need iterative solvers like `GMRES` (Package: IterativeSolvers).

Rank 1 update: Sherman-Morrison formula

$$(A - uv^T)^{-1} = A^{-1} + A^{-1}u(1 - v^T A^{-1} u)^{-1} v^T A^{-1}$$

which requires only $O(n^2)$ extra work.

Assignments

1. Get the relative condition number of division operation a/b.
2. Classify each of the following matrices as well-conditioned or ill-conditioned:

$$(a).\quad \begin{pmatrix} 10^{10} & 0 \\ 0 & 10^{-10} \end{pmatrix}$$

$$(b).\quad \begin{pmatrix} 10^{10} & 0 \\ 0 & 10^{10} \end{pmatrix}$$

$$(c).\quad \begin{pmatrix} 10^{-10} & 0 \\ 0 & 10^{-10} \end{pmatrix}$$

$$(d).\quad \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

3. Implement the Gauss-Jordan elimination algorithm to compute matrix inverse. In the following example, we first create an augmented matrix $(A|I)$. Then we apply the Gauss-Jordan elimination matrices on the left. The final result is stored in the augmented matrix as (I, A^{-1}).

$$\begin{bmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 4 & 4 & 2 & 0 & 1 & 0 \\ 4 & 6 & 4 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -4 & -6 & -4 & 1 & 0 \\ 0 & -2 & -4 & -4 & 0 & 1 \end{array}\right],$$

$$\begin{bmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & -0.5 & 1 \end{bmatrix} \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -4 & -6 & -4 & 1 & 0 \\ 0 & -2 & -4 & -4 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & -1 & -1 & 0.5 & 0 \\ 0 & -4 & -6 & -4 & 1 & 0 \\ 0 & 0 & -1 & -2 & -0.5 & 1 \end{array}\right],$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|ccc} 1 & 0 & -1 & -1 & 0.5 & 0 \\ 0 & -4 & -6 & -4 & 1 & 0 \\ 0 & 0 & -1 & -2 & -0.5 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & -1 \\ 0 & -4 & 0 & 8 & 4 & -6 \\ 0 & 0 & -1 & -2 & -0.5 & 1 \end{array}\right],$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.25 & 0 \\ 0 & 0 & -1 \end{bmatrix} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & -1 \\ 0 & -4 & 0 & 8 & 4 & -6 \\ 0 & 0 & -1 & -2 & -0.5 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 1 & 0 & -2 & -1 & 1.5 \\ 0 & 0 & 1 & 2 & 0.5 & -1 \end{array}\right],$$

Task: Please implement a function `gauss_jordan` that computes the inverse for a matrix at any size. Please also include the following test in your submission.

```julia
@testset "Gauss Jordan" begin
    n = 10
    A = randn(n, n)
```

```
    @test gauss_jordan(A) * A ≈ Matrix{Float64}(I, n, n)
end
```