Table of Contents

present

Review: Solving linear equations

Given $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$, find $x \in \mathbb{R}^n$ s.t.

$$Ax = b$$

1. LU factorization with Gaussian Elimination (with Pivoting)
2. Sensitivity analysis: Condition number
3. Computing matrix inverse with Guass-Jordan Elimination

Linear Least Square Problem

Data Fitting

Given m data points (t_i, y_i), we wish to find the n-vector x of parameters that gives the "best fit" to the data by the model function $f(t, x)$, with

$$f : \mathbb{R}^{n+1} \to \mathbb{R}$$

$$\min_x \sum_{i=1}^{m} (y_i - f(t_i, x))^2$$

Example

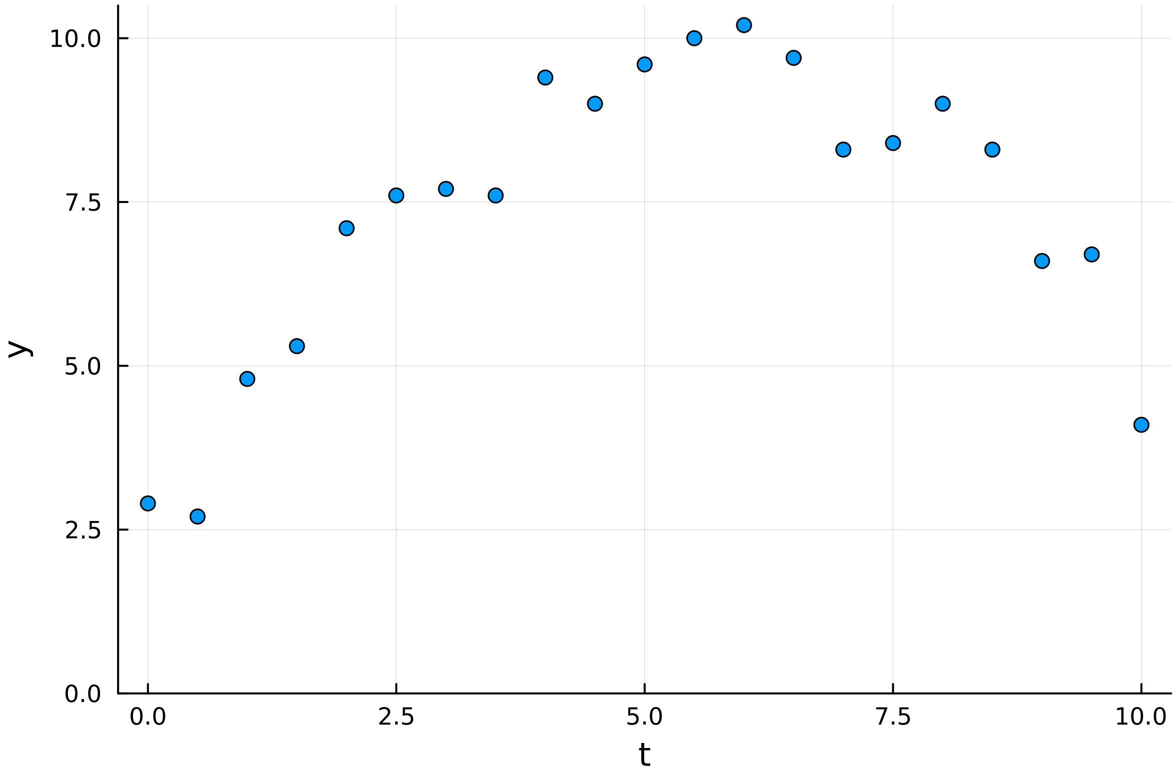
```
ts =
 [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9
```

```julia
ts = collect(0.0:0.5:10.0)
```

```
ys =
 [2.9, 2.7, 4.8, 5.3, 7.1, 7.6, 7.7, 7.6, 9.4, 9.0, 9.6, 10.0, 10.2, 9.7, 8.3, 8.4, 9.0, 8.3,
```

```julia
ys = [2.9, 2.7, 4.8, 5.3, 7.1, 7.6, 7.7, 7.6, 9.4, 9.0, 9.6, 10.0, 10.2, 9.7, 8.3,
8.4, 9.0, 8.3, 6.6, 6.7, 4.1]
```

```julia
using Plots
```




```
scatter(ts, ys; label="", xlabel="t", ylabel="y", ylim=(0, 10.5))
```

$$f(x) = x_0 + x_1 t + x_2 t^2$$

$$Ax = \begin{pmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \approx \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \end{pmatrix} = b$$

```
A2 = 21×3 Matrix{Float64}:
      1.0   0.0    0.0
      1.0   0.5    0.25
      1.0   1.0    1.0
      1.0   1.5    2.25
      1.0   2.0    4.0
      1.0   2.5    6.25
      1.0   3.0    9.0
      ⋮
      1.0   7.5   56.25
      1.0   8.0   64.0
      1.0   8.5   72.25
      1.0   9.0   81.0
      1.0   9.5   90.25
      1.0  10.0  100.0
```

```
A2 = [ones(length(ts)) ts ts.^2]
```

Normal Equations

The goal: minimize $\|Ax - b\|_2^2$

$$A^T A x = A^T b$$

Pseudo-Inverse

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

$$x = A^+ b$$

```
21×3 Matrix{Float64}:
 1.0   0.0    0.0
 1.0   0.5    0.25
 1.0   1.0    1.0
 1.0   1.5    2.25
 1.0   2.0    4.0
 1.0   2.5    6.25
 1.0   3.0    9.0
 ⋮
 1.0   7.5   56.25
 1.0   8.0   64.0
 1.0   8.5   72.25
 1.0   9.0   81.0
 1.0   9.5   90.25
 1.0  10.0  100.0
```

```
A2
```

Pseudoinverse

```
3×21 Matrix{Float64}:
  0.356296    0.289667     0.228684    …   0.0208922    0.0559006    0.0965556
 -0.138905   -0.102428    -0.0695177      -0.0279592   -0.0556748   -0.0869565
  0.0112931   0.00790514   0.00487384      0.00487384   0.00790514   0.0112931
```

```
inv(A2' * A2) * A2'
```

The julia version

```
A2inv = 3×21 Matrix{Float64}:
          0.356296    0.289667     0.228684    …   0.0208922    0.0559006    0.0965556
         -0.138905   -0.102428    -0.0695177      -0.0279592   -0.0556748   -0.0869565
          0.0112931   0.00790514   0.00487384      0.00487384   0.00790514   0.0112931
```

```
A2inv = pinv(A2)
```

Example

```
3×3 Matrix{Float64}:
  21.0    105.0     717.5
 105.0    717.5    5512.5
 717.5   5512.5   45166.6
```

```
A2' * A2
```

```
[155.0, 830.05, 5512.02]
```

```
A2' * ys
```

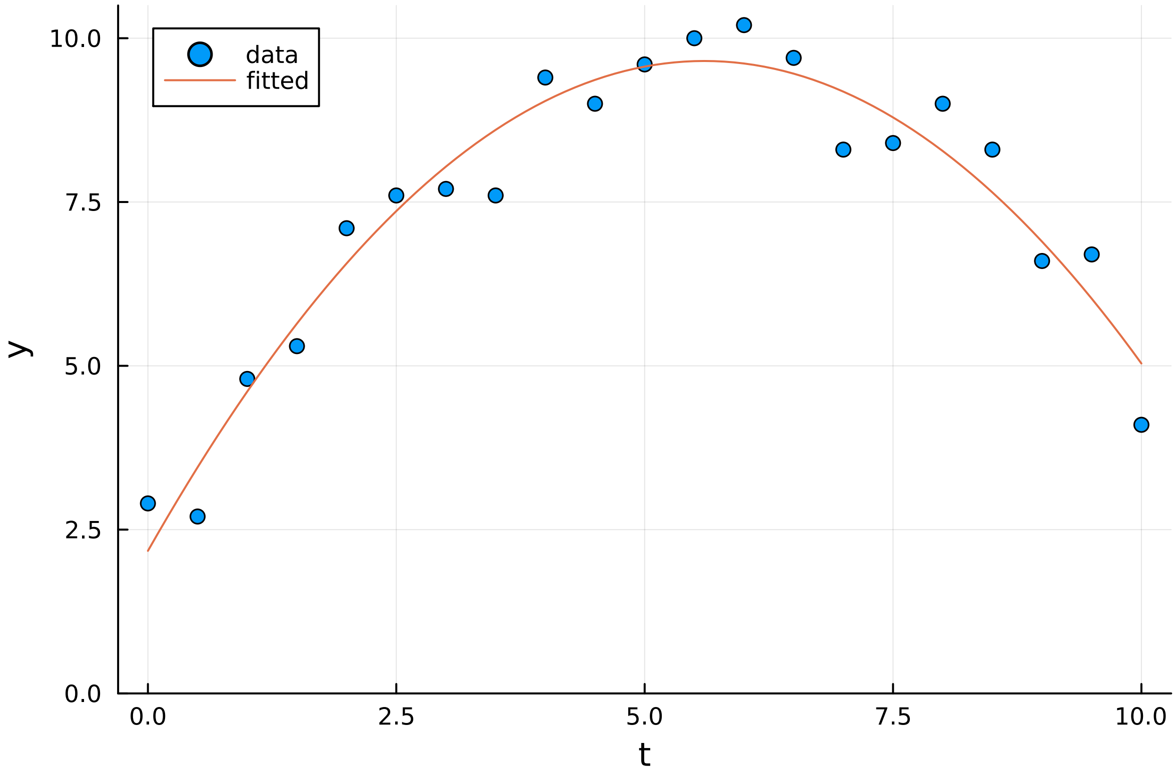
```
x2 =  [2.17572, 2.67041, -0.238444]
```

```
x2 = pinv(A2) * ys
```

```
using LinearAlgebra
```

```
6.795716009391075
```

```
norm(A2 * x2 - ys)^2
```




```
let
    plt = scatter(ts, ys; xlabel="t", ylabel="y", ylim=(0, 10.5), label="data")
    tt = 0:0.1:10
    plot!(plt, tt, map(t->x2[1] + x2[2]*t + x2[3] * t^2, tt); label="fitted")
end
```

The geometric interpretation

The residual is $b - Ax$

$$A^T(b - Ax) = 0$$

Solving Normal Equations with Cholesky decomposition

Step 1: Rectangular ⟶ Square

$$A^T Ax = A^T b$$

Step 2: Square ⟶ Triangular

$$A^T A = LL^T$$

Step 3: Solve the triangular linear equation

Issue: The Condition-Squaring Effect

The conditioning of a square linear system $Ax = b$ depends only on the matrix, while the conditioning of a least squares problem $Ax \approx b$ depends on both A and b.

$$A = \begin{pmatrix} 1 & 1 \\ \epsilon & 0 \\ 0 & \epsilon \end{pmatrix}$$

```
3×21 Matrix{Float64}:
  0.356296    0.289667     0.228684   …   0.0208922    0.0559006    0.0965556
 -0.138905   -0.102428    -0.0695177     -0.0279592   -0.0556748   -0.0869565
  0.0112931   0.00790514   0.00487384     0.00487384   0.00790514   0.0112931
```

```
pinv(A2)
```

```
137.77116637433434
```

```
cond(A2)
```

The definition of thin matrix condition number

```
137.77116637433443
```

```
opnorm(A2) * opnorm(pinv(A2))
```

```
137.77116637433437
```

```
maximum(svd(A2).S)/minimum(svd(A2).S)
```

The algorithm matters

$$x^2 - 2px - q$$

Algorithm 1:

$$p - \sqrt{p^2 + q}$$

Algorithm 2:

$$\frac{q}{p + \sqrt{p^2 + q}}$$

```
-4.0978193283081055e-8
```

```
let
    p = 12345678
    q = 1
    p - sqrt(p^2 + q)
end
```

```
4.0500003321000205e-8
```

```
let # more accurate
    p = 12345678
    q = 1
    q/(p + sqrt(p^2 + q))
end
```

Orthogonal Transformations

$$A = QR$$

$$Rx = Q^T b$$

```
rectQ = 21×3 Matrix{Float64}:
         -0.218218  -0.360375  -0.422855
         -0.218218  -0.324337  -0.295999
         -0.218218  -0.2883    -0.182495
         -0.218218  -0.252262  -0.0823455
         -0.218218  -0.216225   0.00445111
         -0.218218  -0.180187   0.0778944
         -0.218218  -0.14415    0.137984
           ⋮
         -0.218218   0.180187   0.0778944
         -0.218218   0.216225   0.00445111
         -0.218218   0.252262  -0.0823455
         -0.218218   0.2883    -0.182495
         -0.218218   0.324337  -0.295999
         -0.218218   0.360375  -0.422855
```

```
rectQ = Matrix(qr(A2).Q)
```

```
3×3 Matrix{Float64}:
  1.0           -1.33086e-16  -1.32421e-16
 -1.33086e-16   1.0           -8.7499e-17
 -1.32421e-16  -8.7499e-17     1.0
```

```
rectQ' * rectQ
```

```
3×3 Matrix{Float64}:
 -4.58258  -22.9129  -156.571
  0.0       13.8744   138.744
  0.0        0.0      -37.4438
```

```
qr(A2).R
```

```
true
```

```
rectQ * qr(A2).R ≈ A2
```

Gist of QR factoriaztion by Householder reflection.

Let H_k be an orthogonal matrix, i.e. $H_k^T H_k = I$

$$H_n \ldots H_2 H_1 A = R$$

$$Q = H_1^T H_2^T \ldots H_n^T$$

Review of Elimentary Elimination Matrix

$$M_k = I_n - \tau e_k^T$$

$$\tau = (0, \ldots, 0, \tau_{k+1}, \ldots, \tau_n)^T, \quad \tau_i = \frac{v_i}{v_k}.$$

Keys:

- Gaussian elimination is a recursive algorithm.

elementary_elimination_matrix_1 (generic function with 1 method)

```
function elementary_elimination_matrix_1(A::AbstractMatrix{T}) where T
    n = size(A, 1)
    # create Elementary Elimination Matrices
    M = Matrix{Float64}(I, n, n)
    for i=2:n
        M[i, 1] =  -A[i, 1] ./ A[1, 1]
    end
    return M
end
```

lufact_naive_recur! (generic function with 1 method)

```
function lufact_naive_recur!(L, A::AbstractMatrix{T}) where T
    n = size(A, 1)
    if n == 1
        return L, A
    else
        # eliminate the first column
        m = elementary_elimination_matrix_1(A)
        L .= L * inv(m)
        A .= m * A
        # recurse
        lufact_naive_recur!(view(L, 2:n, 2:n), view(A, 2:n, 2:n))
    end
    return L, A
end
```

true

```
let
    A = [1 2 2; 4 4 2; 4 6 4]
    L = Matrix{Float64}(I, 3, 3)
    R = copy(A)
    lufact_naive_recur!(L, R)
    L * R ≈ A
end
```

Householder reflection

Let $v \in \mathbb{R}^m$ be nonzero, An m-by-m matrix P of the form

$$P = 1 - \beta v v^T, \quad \beta = \frac{2}{v^T v}$$

is a Householder reflection.

(the picture of householder reflection)

Properties of Householder reflection

Householder reflection is symmetric and orthogonal.

```
using Test
```

```
DefaultTestSet("householder property", [], 3, false, false, true, 1.677916937952985e9, 1.6
```

```
@testset "householder property" begin
    v = randn(3)
    β = 2/norm(v, 2)^2
    H = I - β * v * v'
    # symmetric
    @test H' ≈ H
    # reflexive
    @test H^2 ≈ I
    # orthogonal
    @test H' * H ≈ I
end
```

```
Test Summary:        | Pass  Total  Time
householder property |    3      3  0.0s
```

```
struct HouseholderMatrix{T} <: AbstractArray{T, 2}
    v::Vector{T}
    β::T
end
```

```
Base.size(A::HouseholderMatrix) = (length(A.v), length(A.v))
```

```
Base.size(A::HouseholderMatrix, i::Int) = i == 1 || i == 2 ? length(A.v) : 1
```

```
left_mul! (generic function with 2 methods)
```

```
# the 'mul!' interfaces can take two extra factors.
function left_mul!(B, A::HouseholderMatrix)
    B .-= (A.β .* A.v) * (A.v' * B)
    return B
end
```

```
right_mul! (generic function with 2 methods)
```

```
# the 'mul!' interfaces can take two extra factors.
function right_mul!(A, B::HouseholderMatrix)
    A .= A .- (A * (B.β .* B.v)) * B.v'
    return A
end
```

```
# some other methods to avoid ambiguity error
```

```
Base.inv(A::HouseholderMatrix) = A
```

```
Base.adjoint(A::HouseholderMatrix) = A
```

```julia
Base.getindex(A::HouseholderMatrix, i::Int, j::Int) = A.β * A.v[i] * conj(A.v[j])
```

Project a vector to e_1

$$Px = \beta e_1$$

$$v = x \pm \|x\|_2 e_1$$

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

```julia
function householder_matrix(v::AbstractVector{T}) where T
    v = copy(v)
    v[1] -= norm(v, 2)
    return HouseholderMatrix(v, 2/norm(v, 2)^2)
end
```

```
3×3 Matrix{Float64}:
 5.74456       7.31126    4.52602
 8.88178e-16  -0.477767  -0.129612
 8.88178e-16   1.52223    1.87039
```

```julia
let
    A = Float64[1 2 2; 4 4 2; 4 6 4]
    hm = householder_matrix(view(A,:,1))
    hm * A
end
```

Triangular Least Squares Problems

QR Factoriaztion

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

```julia
function householder_qr!(Q::AbstractMatrix{T}, a::AbstractMatrix{T}) where T
    m, n = size(a)
    @assert size(Q, 2) == m
    if m == 1
        return Q, a
    else
        # apply householder matrix
        H = householder_matrix(view(a, :, 1))
        left_mul!(a, H)
        # update Q matrix
        right_mul!(Q, H')
        # recurse
        householder_qr!(view(Q, 1:m, 2:m), view(a, 2:m, 2:n))
    end
    return Q, a
end
```

```
DefaultTestSet("householder QR", [], 2, false, false, true, 1.678113744089609e9, 1.6781137
```

```julia
@testset "householder QR" begin
    A = randn(3, 3)
    Q = Matrix{Float64}(I, 3, 3)
    R = copy(A)
    householder_qr!(Q, R)
    @info R
    @test Q * R ≈ A
    @test Q' * Q ≈ I
end
```

```
3×3 Matrix{Float64}:
  2.92512      1.30579  -1.29249
 -2.22045e-16  1.38135   0.530463
  4.44089e-16  0.0      -0.640345
```

```
Test Summary:  | Pass  Total  Time
householder QR |    2      2  0.3s
```

Givens Rotations

```julia
using Luxor
```

```
[ Info: SnoopPrecompile is analyzing Luxor.jl code...
```

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

```julia
function draw_vectors(initial_vector, final_vector, angle)
    @drawsvg begin
        origin()
        circle(0, 0, 100, :stroke)
        setcolor("gray")
        a, b = initial_vector
        Luxor.arrow(Point(0, 0), Point(a, -b) * 100)
        setcolor("black")
        c, d = final_vector
        Luxor.arrow(Point(0, 0), Point(c, -d) * 100)
        Luxor.text("θ = $angle", 0, 50; valign=:center, halign=:center)
    end 600 400
end
```

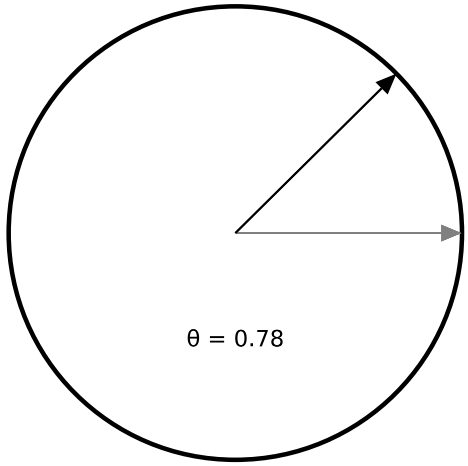
0.0

```julia
@bind angle Slider(0:0.03:2*3.14; show_value=true)
```

$$G = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

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

```julia
rotation_matrix(angle) = [cos(angle) -sin(angle); sin(angle) cos(angle)]
```



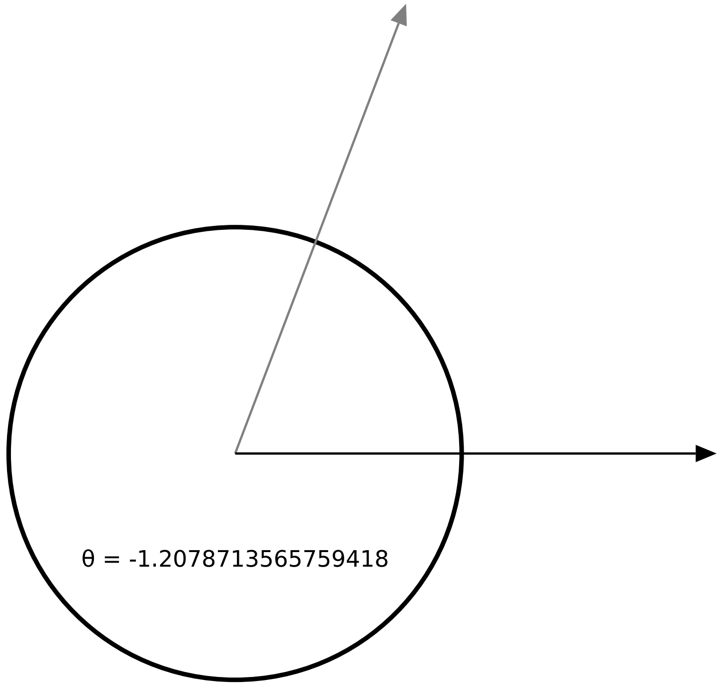

```
let
    initial_vector = [1.0, 0.0]
    final_vector = rotation_matrix(angle) * initial_vector
    @info final_vector
    draw_vectors(initial_vector, final_vector, angle)
end
```

[0.710914, 0.703279]

Eliminating the y element

0.19739555984988078

```
atan(0.1, 0.5)
```




```
let
    initial_vector = randn(2)
    angle = atan(initial_vector[2], initial_vector[1])
    final_vector = rotation_matrix(-angle) * initial_vector
    draw_vectors(initial_vector, final_vector, -angle)
end
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} a_1 \\ \alpha \\ a_3 \\ 0 \\ a_5 \end{pmatrix}$$

where $s = \sin(\theta)$ and $c = \cos(\theta)$.

Givens QR Factorization

```julia
struct GivensMatrix{T} <: AbstractArray{T, 2}
    c::T
    s::T
    i::Int
    j::Int
    n::Int
end
```

```julia
Base.size(g::GivensMatrix) = (g.n, g.n)
```

```julia
Base.size(g::GivensMatrix, i::Int) = i == 1 || i == 2 ? g.n : 1
```

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

```julia
function givens(A, i, j)
    x, y = A[i, 1], A[j, 1]
    norm = sqrt(x^2 + y^2)
    c = x/norm
    s = y/norm
    return GivensMatrix(c, s, i, j, size(A, 1))
end
```

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

```julia
function left_mul!(A::AbstractMatrix, givens::GivensMatrix)
    for col in 1:size(A, 2)
        vi, vj = A[givens.i, col], A[givens.j, col]
        A[givens.i, col] = vi * givens.c + vj * givens.s
        A[givens.j, col] = -vi * givens.s + vj * givens.c
    end
    return A
end
```

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

```julia
function right_mul!(A::AbstractMatrix, givens::GivensMatrix)
    for row in 1:size(A, 1)
        vi, vj = A[row, givens.i], A[row, givens.j]
        A[row, givens.i] = vi * givens.c + vj * givens.s
        A[row, givens.j] = -vi * givens.s + vj * givens.c
    end
    return A
end
```

```
3×3 Matrix{Float64}:
 0.421165  -0.499439   0.0337569
 0.28626    0.959619   0.934689
 0.0       -1.04783   -0.434808
```

```julia
let
    A = randn(3, 3)
    g = givens(A, 2, 3)
    left_mul!(copy(A), g)
end
```

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

```
function givens_qr!(Q::AbstractMatrix, A::AbstractMatrix)
    m, n = size(A)
    if m == 1
        return Q, A
    else
        for k = m:-1:2
            g = givens(A, k-1, k)
            left_mul!(A, g)
            right_mul!(Q, g)
        end
        givens_qr!(view(Q, :, 2:m), view(A, 2:m, 2:n))
        return Q, A
    end
end
```

```
DefaultTestSet("givens QR", [], 2, false, false, true, 1.678113709140285e9, 1.678113709224
```

```
@testset "givens QR" begin
    n = 3
    A = randn(n, n)
    R = copy(A)
    Q, R = givens_qr!(Matrix{Float64}(I, n, n), R)
    @test Q * R ≈ A
    @test Q * Q' ≈ I
    @info R
end
```

```
3×3 Matrix{Float64}:
 2.92512   1.30579      -1.29249
 0.0       1.38135       0.530463
 0.0      -1.11022e-16  -0.640345
```

```
Test Summary: | Pass  Total  Time
givens QR     |    2      2  0.1s
```

Gram-Schmidt Orthogonalization

$$q_k = \left(a_k - \sum_{i=1}^{k-1} r_{ik} q_i\right) / r_{kk}$$

Algorithm: Classical Gram-Schmidt Orthogonalization

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

```julia
function classical_gram_schmidt(A::AbstractMatrix{T}) where T
    m, n = size(A)
    Q = zeros(T, m, n)
    R = zeros(T, n, n)
    R[1, 1] = norm(view(A, :, 1))
    Q[:, 1] .= view(A, :, 1) ./ R[1, 1]
    for k = 2:n
        Q[:, k] .= view(A, :, k)
        # project z to span(A[:, 1:k-1])⊥
        for j = 1:k-1
            R[j, k] = view(Q, :, j)' * view(A, :, k)
            Q[:, k] .-= view(Q, :, j) .* R[j, k]
        end
        # normalize the k-th column
        R[k, k] = norm(view(Q, :, k))
        Q[:, k] ./= R[k, k]
    end
    return Q, R
end
```

```
DefaultTestSet("classical GS", [], 2, false, false, true, 1.678098331619859e9, 1.678098331
```

```julia
@testset "classical GS" begin
    n = 10
    A = randn(n, n)
    Q, R = classical_gram_schmidt(A)
    @test Q * R ≈ A
    @test Q * Q' ≈ I
    @info R
end
```

```
10×10 Matrix{Float64}:
 4.05858  1.80419  0.815898  -0.34024   …   0.931039   -0.382864   -0.75112
 0.0      2.78212  1.10654   -0.965912     -0.126994    0.186296   -0.109283
 0.0      0.0      2.62962   -0.481856     -1.18262     2.08893    -1.01857
 0.0      0.0      0.0        2.46999      -0.991887    1.27968    -0.502267
 0.0      0.0      0.0        0.0           0.0169861  -0.981608   -1.19355
 0.0      0.0      0.0        0.0       …  -0.955197    0.0471644   0.993383
 0.0      0.0      0.0        0.0          -2.05969    -1.1144      1.31633
 0.0      0.0      0.0        0.0           1.86435     0.451551   -0.980721
 0.0      0.0      0.0        0.0           0.0         0.94503     2.10059
 0.0      0.0      0.0        0.0           0.0         0.0         0.153036
```

```
Test Summary: | Pass  Total  Time
classical GS  |    2      2  0.0s
```

Algorithm: Modified Gram-Schmidt Orthogonalization

modified_gram_schmidt! (generic function with 1 method)

```julia
function modified_gram_schmidt!(A::AbstractMatrix{T}) where T
    m, n = size(A)
    Q = zeros(T, m, n)
    R = zeros(T, n, n)
    for k = 1:n
        R[k, k] = norm(view(A, :, k))
        Q[:, k] .= view(A, :, k) ./ R[k, k]
        for j = k+1:n
            R[k, j] = view(Q, :, k)' * view(A, :, j)
            A[:, j] .-= view(Q, :, k) .* R[k, j]
        end
    end
    return Q, R
end
```

DefaultTestSet("modified GS", [], 2, false, false, true, 1.677962674800716e9, 1.6779626748

```julia
@testset "modified GS" begin
    n = 10
    A = randn(n, n)
    Q, R = modified_gram_schmidt!(copy(A))
    @test Q * R ≈ A
    @test Q * Q' ≈ I
    @info R
end
```

```
10×10 Matrix{Float64}:
 4.05858  1.80419  0.815898  -0.34024   …   0.931039   -0.382864   -0.75112
 0.0      2.78212  1.10654   -0.965912     -0.126994    0.186296   -0.109283
 0.0      0.0      2.62962   -0.481856     -1.18262     2.08893    -1.01857
 0.0      0.0      0.0        2.46999      -0.991887    1.27968    -0.502267
 0.0      0.0      0.0        0.0           0.0169861  -0.981608   -1.19355
 0.0      0.0      0.0        0.0       …  -0.955197    0.0471644   0.993383
 0.0      0.0      0.0        0.0          -2.05969    -1.1144      1.31633
 0.0      0.0      0.0        0.0           1.86435     0.451551   -0.980721
 0.0      0.0      0.0        0.0           0.0         0.94503     2.10059
 0.0      0.0      0.0        0.0           0.0         0.0         0.153036
```

```
Test Summary: | Pass  Total  Time
modified GS   |    2      2  0.0s
```

```julia
let
    n = 100
    A = randn(n, n)
    Q1, R1 = classical_gram_schmidt(A)
    Q2, R2 = modified_gram_schmidt!(copy(A))
    @info norm(Q1' * Q1 - I)
    @info norm(Q2' * Q2 - I)
end
```

6.993469646172434e-13

1.5592036166435736e-13

Eigenvalue/Singular value decomposition problem

$$Ax = \lambda x$$

Power method

```
matsize = 10
```

```julia
matsize = 10
```

```
10×10 Matrix{Float64}:
 -0.131984     0.43241   -0.919488  -0.300586   …   0.384314   -0.00511473  -0.547111
  0.43241      1.70421   -1.04985   -0.632086       1.35511     2.3797      -1.85416
 -0.919488    -1.04985   -0.168887  -1.26338       -1.04436    -3.92819      0.748691
 -0.300586    -0.632086  -1.26338   -0.0828375      0.827814   -0.546925    -1.06453
 -0.978485    -0.455107  -4.32425   -0.0165129     -1.59179     1.18557     -0.497179
  0.372636     2.90987   -1.68008   -0.207393   …   0.0605694   0.689852    -0.829824
 -0.800089     1.21472   -2.17438   -1.52431        1.94168     1.69848     -0.806781
  0.384314     1.35511   -1.04436    0.827814      -1.9143     -1.54797     -0.586512
 -0.00511473   2.3797    -3.92819   -0.546925      -1.54797     2.31972     -2.45903
 -0.547111    -1.85416    0.748691  -1.06453       -0.586512   -2.45903     -0.292873
```

```julia
A10 = randn(matsize, matsize); A10 += A10'
```

```
[-7.01061, -4.31622, -2.25874, -1.82846, -0.0268208, 0.828659, 1.36902, 2.95641, 4.3959, 1
```

```julia
eigen(A10).values
```

```
vmax =
 [-0.0444174, -0.412083, 0.466506, 0.0128633, -0.214195, -0.321107, -0.282626, -0.0491481,
```

```julia
vmax = eigen(A10).vectors[:,end]
```

```
1.6032776772867408e-8
```

```julia
let
    x = normalize!(randn(matsize))
    for i=1:20
        x = A10 * x
        normalize!(x)
    end
    1-abs2(x' * vmax)
end
```

Rayleigh Quotient Iteration

```
[-1.51268e-15, 2.97123e-14, 6.60583e-13, 2.92461e-12, -6.31891e-11, -3.91916e-8, -1.0, 2.3
```

```
let
    x = normalize!(randn(matsize))
    U = eigen(A10).vectors
    for k=1:5
        sigma = x' * A10 * x
        y = (A10 - sigma * I) \ x
        x = normalize!(y)
    end
    (x' * U)'
end
```

Symmetric QR decomposition

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

```
function householder_trid!(Q, a)
    m, n = size(a)
    @assert m==n && size(Q, 2) == n
    if m == 2
        return Q, a
    else
        # apply householder matrix
        H = householder_matrix(view(a, 2:n, 1))
        left_mul!(view(a, 2:n, :), H)
        right_mul!(view(a, :, 2:n), H')
        # update Q matrix
        right_mul!(view(Q, :, 2:n), H')
        # recurse
        householder_trid!(view(Q, :, 2:n), view(a, 2:m, 2:n))
    end
    return Q, a
end
```

```
DefaultTestSet("householder tridiagonal", [], 1, false, false, true, 1.678115332973689e9,
```

```
@testset "householder tridiagonal" begin
    n = 5
    a = randn(n, n)
    a = a + a'
    Q = Matrix{Float64}(I, n, n)
    Q, T = householder_trid!(Q, copy(a))
    @test Q * T * Q' ≈ a
end
```

```
Test Summary:          | Pass  Total  Time
householder tridiagonal |    1      1  0.0s
```

The SVD algorithm

$$A = USV^T$$

1. Form $C = A^T A$,
2. Use the symmetric QR algorithm to compute $V_1^T C V_1 = \mathbf{diag}(\sigma_i^2)$,
3. Apply QR with column pivoting to AV_1 obtaining $U^T(AV_1)\Pi = R$.

Assignments

1. Review

Suppose that you are computing the QR factorization of the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{pmatrix}$$

by Householder transformations.

- Problems:
  1. How many Householder transformations are required?
  2. What does the first column of A become as a result of applying the first Householder transformation?
  3. What does the first column of A become as a result of applying the first Householder transformation?
  4. How many Givens rotations would be required to computing the QR factoriazation of A?

2. Coding

Computing the QR decomposition of a symmetric triangular matrix with Givens rotation. Try to minimize the computing time and estimate the number of FLOPS.

For example, if the input matrix size is $T \in \mathbb{R}^{5\times 5}$

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & 0 \\ 0 & 0 & t_{43} & t_{44} & t_{45} \\ 0 & 0 & 0 & t_{54} & t_{55} \end{pmatrix}$$

where $t_{ij} = t_{ji}$.

In your algorithm, you should first apply Givens rotation on row 1 and 2.

$$G(t_{11}, t_{21})T = \begin{pmatrix} t'_{11} & t'_{12} & t'_{13} & 0 & 0 \\ 0 & t'_{22} & t'_{23} & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & 0 \\ 0 & 0 & t_{43} & t_{44} & t_{45} \\ 0 & 0 & 0 & t_{54} & t_{55} \end{pmatrix}$$

Then apply $G(t'_{22}, t_{32})$ et al.