```julia
using PlutoUI
```

Table of Contents

present

Overview

1. Sparse matrix representation.
    - COOrdinate (COO) format
    - Compressed Sparse Column/Row (CSC/CSR) format
2. Solving the dominant eigenvalue problem.
    - Symmetric Lanczos process
    - Anoldi process

Sparse Matrices

```
using LinearAlgebra    , SparseArrays
```

Recall that the elementary elimination matrix in Gaussian elimination has the following form.

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

where $m_i = a_i/a_k$.

The following cell is copied from notebook: `4.linearequation.jl`

`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
```

```
some_random_matrix = 5×5 reshape(::UnitRange{Int64}, 5, 5) with eltype Int64:
                      1   6  11  16  21
                      2   7  12  17  22
                      3   8  13  18  23
                      4   9  14  19  24
                      5  10  15  20  25
```

```julia
some_random_matrix = reshape(1:25, 5, 5)
```

```
demo_matrix = 5×5 Matrix{Float64}:
               1.0  0.0   0.0      0.0  0.0
               0.0  1.0   0.0      0.0  0.0
               0.0  0.0   1.0      0.0  0.0
               0.0  0.0  -1.07692  1.0  0.0
               0.0  0.0  -1.15385  0.0  1.0
```

```julia
demo_matrix = elementary_elimination_matrix(some_random_matrix, 3)
```

This representation requires storing n^2 elements, which is very memory inefficient since it has only $2n - k$ nonzero elements.

Let $A \in \mathbb{R}^{m\times n}$ be a sparse matrix, and $\mathbf{nnz}(A) \ll mn$ be the number of nonzero elements in A. Is there a universal matrix type that stores such sparse matrices efficiently?.

The answer is yes.

COOrdinate (COO) format

The coordinate format means storing nonzero matrix elements into triples

$$\begin{aligned}&(i_1, j_1, v_1)\\&(i_2, j_2, v_2)\\&\vdots\\&(i_k, j_k, v_k)\end{aligned}$$

Quiz: How many bytes are required to store the matrix `demo_matrix` in the COO format?

```julia
struct COOMatrix{T} <: AbstractArray{T, 2}   # Julia does not have a COO data type
    rowval::Vector{Int}   # row indices
    colval::Vector{Int}   # column indices
    nzval::Vector{T}      # values
    m::Int                # number of rows
    n::Int                # number of columns
end
```

We need to implement the `AbstractArray` interfaces.

```julia
Base.size(coo::COOMatrix{T}) where T = (coo.m, coo.n)
Base.size(coo::COOMatrix{T}, i::Int) where T = getindex((coo.m, coo.n), i)
```

Indexing a COO matrix

Element indexing requires $O(\mathrm{nnz}(A))$ time.

```julia
function Base.getindex(coo::COOMatrix{T}, i::Integer, j::Integer) where T
    v = zero(T)
    for (i2, j2, v2) in zip(coo.rowval, coo.colval, coo.nzval)
        if i == i2 && j == j2
            v += v2  # accumulate the value, since repeated indices are allowed.
        end
    end
    return v
end
```

```
coo_matrix = 5×5 COOMatrix{Float64}:
              1.0  0.0   0.0      0.0  0.0
              0.0  1.0   0.0      0.0  0.0
              0.0  0.0   1.0      0.0  0.0
              0.0  0.0  -1.07692  1.0  0.0
              0.0  0.0  -1.15385  0.0  1.0
```

```julia
coo_matrix = COOMatrix([1, 2, 3, 4, 5, 4, 5], [1, 2, 3, 4, 5, 3, 3], [1, 1, 1, 1, 1,
demo_matrix[4,3], demo_matrix[5, 3]], 5, 5)
```

```julia
# uncomment to show the result
# sizeof(coo_format)
```

Multiplying two COO matrices

In the following example, we compute `coo_matrix * coo_matrix`.

```julia
function Base.:(*)(A::COOMatrix{T1}, B::COOMatrix{T2}) where {T1, T2}
    @assert size(A, 2) == size(B, 1)
    rowval = Int[]
    colval = Int[]
    nzval = promote_type(T1, T2)[]
    for (i, j, v) in zip(A.rowval, A.colval, A.nzval)
        for (i2, j2, v2) in zip(B.rowval, B.colval, B.nzval)
            if j == i2
                push!(rowval, i)
                push!(colval, j2)
                push!(nzval, v * v2)
            end
        end
    end
    return COOMatrix(rowval, colval, nzval, size(A, 1), size(B, 2))
end
```

```
5×5 COOMatrix{Float64}:
 1.0  0.0   0.0       0.0  0.0
 0.0  1.0   0.0       0.0  0.0
 0.0  0.0   1.0       0.0  0.0
 0.0  0.0  -2.15385   1.0  0.0
 0.0  0.0  -2.30769   0.0  1.0
```

```julia
coo_matrix * coo_matrix
```

```
5×5 Matrix{Float64}:
 1.0  0.0   0.0       0.0  0.0
 0.0  1.0   0.0       0.0  0.0
 0.0  0.0   1.0       0.0  0.0
 0.0  0.0  -2.15385   1.0  0.0
 0.0  0.0  -2.30769   0.0  1.0
```

```julia
demo_matrix ^ 2
```

Yep!

Quiz: What is the time complexity of COO matrix multiplication?

Compressed Sparse Column (CSC) format

A CSC format sparse matrix can be constructed with the `SparseArrays.sparse` function

```
csc_matrix = 5×5 SparseMatrixCSC{Float64, Int64} with 7 stored entries:
              1.0   ⋅     ⋅         ⋅    ⋅
               ⋅   1.0    ⋅         ⋅    ⋅
               ⋅    ⋅    1.0        ⋅    ⋅
               ⋅    ⋅   -1.07692   1.0   ⋅
               ⋅    ⋅   -1.15385    ⋅   1.0
```

```julia
csc_matrix = sparse(coo_matrix.rowval, coo_matrix.colval, coo_matrix.nzval)
```

It contains 5 fields

```
(:m, :n, :colptr, :rowval, :nzval)
```

```
fieldnames(csc_matrix |> typeof)
```

The `m`, `n`, `rowval` and `nzval` have the same meaning as those in the COO format. `colptr` is a integer vector of size $n+1$, the element of which points to the elements in `rowval` and `nzval`. Given a matrix $A \in \mathbb{R}^{m\times n}$ in the CSC format, the j-th column of A is defined as `A[rowval[colptr[j]:colptr[j+1]-1], j] := rowval[colptr[j]:colptr[j+1]-1]`

```
SparseArrays.SparseVector{Float64, Int64}: [0.0, 0.0, 1.0, -1.07692, -1.15385]
```

```
csc_matrix[:, 3]
```

The row indices of nonzero elements in the 3rd column.

```
rows3 =  [3, 4, 5]
```

```
rows3 = csc_matrix.rowval[csc_matrix.colptr[3]:csc_matrix.colptr[4]-1]
```

```
[3, 4, 5]
```

```
# or equivalently in Julia, we can use 'nzrange'
csc_matrix.rowval[nzrange(csc_matrix, 3)]
```

The values of nonzero elements in the 3rd column.

```
[1.0, -1.07692, -1.15385]
```

```
csc_matrix.nzval[csc_matrix.colptr[3]:csc_matrix.colptr[4]-1]
```

Indexing a CSC matrix

The number of operations required to index an element in the j-th column of a CSC matrix is linear to the nonzero elements in the j-th column.

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

```
# I do not want to overwrite 'Base.getindex'
function my_getindex(A::SparseMatrixCSC{T}, i::Int, j::Int) where T
    for k in nzrange(A, j)
        if A.rowval[k] == i
            return A.nzval[k]
        end
    end
    return zero(T)
end
```

```
-1.0769230769230769
```

```
my_getindex(csc_matrix, 4, 3)
```

Multiplying two CSC matrices

Multiplying two CSC matrices is much faster than multiplying two COO matrices.

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

```julia
function my_matmul(A::SparseMatrixCSC{T1}, B::SparseMatrixCSC{T2}) where {T1, T2}
    T = promote_type(T1, T2)
    @assert size(A, 2) == size(B, 1)
    rowval, colval, nzval = Int[], Int[], T[]
    for j2 in 1:size(B, 2)  # enumerate the columns of B
        for k2 in nzrange(B, j2)  # enumerate the rows of B
            v2 = B.nzval[k2]
            for k1 in nzrange(A, B.rowval[k2])  # enumerate the rows of A
                push!(rowval, A.rowval[k1])
                push!(colval, j2)
                push!(nzval, A.nzval[k1] * v2)
            end
        end
    end
    return sparse(rowval, colval, nzval, size(A, 1), size(B, 2))
end
```

```
5×5 SparseMatrixCSC{Float64, Int64} with 7 stored entries:
 1.0   ⋅     ⋅         ⋅    ⋅
  ⋅   1.0    ⋅         ⋅    ⋅
  ⋅    ⋅    1.0        ⋅    ⋅
  ⋅    ⋅   -2.15385  1.0    ⋅
  ⋅    ⋅   -2.30769   ⋅   1.0
```

```julia
my_matmul(csc_matrix, csc_matrix)
```

```
5×5 SparseMatrixCSC{Float64, Int64} with 7 stored entries:
 1.0   ⋅     ⋅         ⋅    ⋅
  ⋅   1.0    ⋅         ⋅    ⋅
  ⋅    ⋅    1.0        ⋅    ⋅
  ⋅    ⋅   -2.15385  1.0    ⋅
  ⋅    ⋅   -2.30769   ⋅   1.0
```

```julia
csc_matrix^2
```

Large sparse eigenvalue problem

Dominant eigenvalue problem

One can use the power method to compute dominant eigenvalues (one having the largest absolute value) of a matrix.

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

```
function power_method(A::AbstractMatrix{T}, n::Int) where T
    n = size(A, 2)
    x = normalize!(randn(n))
    for i=1:n
        x = A * x
        normalize!(x)
    end
    return x' * A * x', x
end
```

Since computing matrix-vector multiplication of CSC sparse matrix is fast, the power method is a convenient method to obtain the largest eigen value of a sparse matrix.

The rate of convergence is dedicated by $|\lambda_2/\lambda_1|^k$.

By inverting the sign, $A \rightarrow -A$, we can use the same method to obtain the smallest eigenvalue.

The symmetric Lanczos process

Let $A \in \mathbb{R}^{n\times n}$ be a large symmetric sparse matrix, the Lanczos process can be used to obtain its largest/smallest eigenvalue, with faster convergence speed comparing with the power method.

The Krylov subspace

A Krylov subspace of size k with initial vector q_1 is defined by

$$\mathcal{K}(A, q_1, k) = \text{span}\{q_1, Aq_1, A^2q_1, \ldots, A^{k-1}q_1\}$$

The Julia package `KrylovKit.jl` contains many Krylov space based algorithms.

`KrylovKit.jl` accepts general functions or callable objects as linear maps, and general Julia objects with vector like behavior (as defined in the docs) as vectors.

The high level interface of KrylovKit is provided by the following functions:

- `linsolve`: solve linear systems
- `eigsolve`: find a few eigenvalues and corresponding eigenvectors
- `geneigsolve`: find a few generalized eigenvalues and corresponding vectors
- `svdsolve`: find a few singular values and corresponding left and right singular vectors
- `exponentiate`: apply the exponential of a linear map to a vector
- `expintegrator`: exponential integrator for a linear non-homogeneous ODE, computes a linear combination of the `ϕⱼ` functions which generalize `ϕ₀(z) = exp(z)`.

```julia
using KrylovKit
```

Projecting a sparse matrix into a subspace

Given $Q \in \mathbb{R}^{n\times k}$ and $Q^T Q = I$, the following statement is always true.

$$\lambda_1(Q_k^T A Q) \leq \lambda_1(A),$$

where $\lambda_1(A)$ is the largest eigenvalue of $A \in \mathbb{R}^{n\times n}$.

Lanczos Tridiagonalization

In the Lanczos tridiagonalizaiton process, we want to find a orthogonal matrix Q^T such that

$$Q^T A Q = T$$

where T is a tridiagonal matrix

$$T = \left(\begin{matrix}\alpha_1 & \beta_1 & 0 & \ldots & 0\\ \beta_1 & \alpha_2 & \beta_2 & \ldots & 0\\ 0 & \beta_2 & \alpha_3 & \ldots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \beta_{k-1} & \alpha_k\end{matrix}\right),$$

$Q = [q_1|q_2|\ldots|q_n]$, and $\mathrm{span}(\{q_1, q_2, \ldots, q_k\}) = \mathcal{K}(A, q_1, k)$.

We have $Aq_k = \beta_{k-1}q_{k-1} + \alpha_k q_k + \beta_k q_{k+1}$, or equivalently in the recursive style

$$q_{k+1} = (Aq_k - \beta_{k-1}q_{k-1} - \alpha_k q_k)/\beta_k.$$

By multiplying q_k^T on the left, we have

$$\alpha_k = q_k^T A q_k.$$

Since q_{k+1} is normalized, we have

$$\beta_k = \|Aq_k - \beta_{k-1}q_{k-1} - \alpha_k q_k\|_2$$

If at any moment, $\beta_k = 0$, the interation stops due to convergence of a subspace. We have the following reducible form

$$T(\beta_2 = 0) = \left(\begin{array}{cc|ccc} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ \beta_1 & \alpha_2 & 0 & \dots & 0 \\ \hline 0 & 0 & \alpha_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \beta_{k-1} & \alpha_k \end{array}\right),$$

A naive implementation

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

```
function lanczos(A, q1::AbstractVector{T}; abstol, maxiter) where T
    # normalize the input vector
    q1 = normalize(q1)
    # the first iteration
    q = [q1]
    Aq1 = A * q1
    α = [q1' * Aq1]
    rk = Aq1 .- α[1] .* q1
    β = [norm(rk)]
    for k = 2:min(length(q1), maxiter)
        # the k-th orthonormal vector in Q
        push!(q, rk ./ β[k-1])
        Aqk = A * q[k]
        # compute the diagonal element as αₖ = qₖᵀ A qₖ
        push!(α, q[k]' * Aqk)
        rk = Aqk .- α[k] .* q[k] .- β[k-1] * q[k-1]
        # compute the off-diagonal element as βₖ = |rₖ|
        nrk = norm(rk)
        # break if βₖ is smaller than abstol or the maximum number of iteration is
        reached
        if abs(nrk) < abstol || k == length(q1)
            break
        end
        push!(β, nrk)
    end
    # returns T and Q
    return SymTridiagonal(α, β), hcat(q...)
end
```

Example: using dominant eigensolver to study the spectral graph theory

Laplacian matrix Given a simple graph G with n vertices $v_1, \ldots, v_n$, its Laplacian matrix $L_{n\times n}$ is defined element-wise as

$$L_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise,} \end{cases}$$

or equivalently by the matrix $L = D - A$, where D is the degree matrix and A is the adjacency matrix of the graph. Since G is a simple graph, A only contains 1s or 0s and its diagonal elements are all 0s.

Theorem: The number of connected components in the graph is the dimension of the nullspace of the Laplacian and the algebraic multiplicity of the 0 eigenvalue.

```
• using Graphs      # for generating sparse matrices
```

```
graphsize = 10
```

```
• graphsize = 10
```

One can use the `Graphs.laplacian_matrix(graph)` to generate a laplacian matrix (CSC formated) of a graph.

```
lmat = 10×10 SparseMatrixCSC{Int64, Int64} with 40 stored entries:
         3   ⋅  -1   ⋅   ⋅  -1   ⋅  -1   ⋅   ⋅
         ⋅   3   ⋅  -1  -1   ⋅  -1   ⋅   ⋅   ⋅
        -1   ⋅   3   ⋅   ⋅  -1   ⋅   ⋅  -1   ⋅
         ⋅  -1   ⋅   3  -1   ⋅   ⋅   ⋅   ⋅  -1
         ⋅  -1   ⋅  -1   3   ⋅   ⋅   ⋅  -1   ⋅
        -1   ⋅  -1   ⋅   ⋅   3   ⋅  -1   ⋅   ⋅
         ⋅  -1   ⋅   ⋅   ⋅   ⋅   3   ⋅  -1  -1
        -1   ⋅   ⋅   ⋅   ⋅  -1   ⋅   3   ⋅  -1
         ⋅   ⋅  -1   ⋅  -1   ⋅  -1   ⋅   3   ⋅
         ⋅   ⋅   ⋅  -1   ⋅   ⋅  -1  -1   ⋅   3
```

```
• lmat = laplacian_matrix(random_regular_graph(graphsize, 3))
```

```
 (10×10 SymTridiagonal{Float64, Vector{Float64}}:                                       , 10×10 Ma
  2.72639  1.32199   ⋅        ⋅        …  ⋅         ⋅         ⋅         ⋅          -0.1720
  1.32199  2.07851  1.40471   ⋅           ⋅         ⋅         ⋅         ⋅          -0.0289
   ⋅       1.40471  2.94481  1.6509       ⋅         ⋅         ⋅         ⋅          -0.2171
   ⋅        ⋅       1.6509   3.22164      ⋅         ⋅         ⋅         ⋅          -0.4783
   ⋅        ⋅        ⋅       1.28649      ⋅         ⋅         ⋅         ⋅          -0.0510
   ⋅        ⋅        ⋅        ⋅        …  0.953022  ⋅         ⋅         ⋅           0.4532
   ⋅        ⋅        ⋅        ⋅           3.1485   1.12749    ⋅         ⋅           0.1609
   ⋅        ⋅        ⋅        ⋅           1.12749  3.24724   0.244936   ⋅          0.5905
   ⋅        ⋅        ⋅        ⋅            ⋅       0.244936  4.45314   0.345662    0.1063
   ⋅        ⋅        ⋅        ⋅            ⋅        ⋅        0.345662  3.24229     0.3155
```

```
• tri, Q = lanczos(lmat, randn(graphsize); abstol=1e-8, maxiter=100)
```

```
 [6.21725e-15, 0.561877, 1.75302, 2.27451, 3.1485, 3.44504, 4.0, 4.53499, 4.80194, 5.48011]
```

```
• eigen(tri).values
```

```
10×10 Matrix{Float64}:
  1.0           4.52615e-16   4.39732e-16  …   6.77108e-15  -4.77572e-14
  4.52615e-16   1.0          -5.7249e-16      -3.90572e-15   1.21535e-14
  4.39732e-16  -5.7249e-16    1.0             -9.85671e-15   3.37725e-14
 -6.02034e-16   2.5731e-16    1.01199e-15      1.60404e-15  -9.14246e-14
  8.47583e-17   8.80291e-16   4.57865e-16     -1.07861e-14   1.31437e-13
  1.23857e-15   1.2562e-16   -8.6211e-16   …   1.17106e-14  -1.14332e-13
 -6.91691e-17  -1.20709e-15  -6.54035e-16     -3.33686e-15   2.02581e-14
 -2.5537e-15    2.07489e-16   1.01226e-15     -7.08993e-15   1.39627e-14
  6.77108e-15  -3.90572e-15  -9.85671e-15      1.0           2.27019e-15
 -4.77572e-14   1.21535e-14   3.37725e-14      2.27019e-15   1.0
```

```
• Q' * Q
```

10

```
• @bind graph_size Slider(10:2:200; show_value=true, default=10)
```

```julia
let
    graph = random_regular_graph(graph_size, 3)
    A = laplacian_matrix(graph)
    q1 = randn(graph_size)
    tr, Q = lanczos(-A, q1; abstol=1e-8, maxiter=100)
    # using function `KrylovKit.eigsolve`
    @info "KrylovKit.eigsolve: " eigsolve(A, q1, 2, :SR)
    # diagonalize the triangular matrix obtained with our naive implementation
    @info "Naive approach: " eigen(-tr).values
end;
```

```
KrylovKit.eigsolve:
eigsolve(A, q1, 2, :SR):
    ([7.99361e-15, 0.885092, 1.38197, 2.38197, 3.2541, 3.38197, 3.61803, 4.61803,

Naive approach:
(eigen(-tr)).values:
    [8.88178e-15, 0.885092, 1.38197, 2.38197, 3.2541, 3.38197, 3.61803, 4.61803,
```

NOTE: with larger `graph_size`, you should see some "ghost" eigenvalues

Reorthogonalization

Let $r_0, \ldots, r_{k-1} \in \mathbb{R}_n$ be given and suppose that Householder matrices $H_0, \ldots, H_{k-1}$ have been computed such that $(H_0 \ldots H_{k-1})^T [r_0 \mid \ldots \mid r_{k-1}]$ is upper triangular. Let $[q_1 \mid \ldots \mid q_k]$ denote the first k columns of the Householderproduct $(H_0 \ldots H_{k-1})$. Then $q_k^T q_l = \delta_{kl}$ (machine precision).

The following 4 cells are copied from notebook: 5.linear-least-square.jl

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

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

```julia
# 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 1 method)
```

```
# 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
```

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

```
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
```

The Lanczos algorithm with complete orthogonalization.

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

```julia
function lanczos_reorthogonalize(A, q1::AbstractVector{T}; abstol, maxiter) where T
    n = length(q1)
    # normalize the input vector
    q1 = normalize(q1)
    # the first iteration
    q = [q1]
    Aq1 = A * q1
    α = [q1' * Aq1]
    rk = Aq1 .- α[1] .* q1
    β = [norm(rk)]
    householders = [householder_matrix(q1)]
    for k = 2:min(n, maxiter)
        # reorthogonalize rk: 1. compute the k-th householder matrix
        for j = 1:k-1
            left_mul!(view(rk, j:n), householders[j])
        end
        push!(householders, householder_matrix(view(rk, k:n)))
        # reorthogonalize rk: 2. compute the k-th orthonormal vector in Q
        qk = zeros(T, n); qk[k] = 1  # qₖ = H₁H₂…Hₖeₖ
        for j = k:-1:1
            left_mul!(view(qk, j:n), householders[j])
        end
        push!(q, qk)
        Aqk = A * q[k]
        # compute the diagonal element as αₖ = qₖᵀ A qₖ
        push!(α, q[k]' * Aqk)
        rk = Aqk .- α[k] .* q[k] .- β[k-1] * q[k-1]
        # compute the off-diagonal element as βₖ = |rₖ|
        nrk = norm(rk)
        # break if βₖ is smaller than abstol or the maximum number of iteration is
        reached
        if abs(nrk) < abstol || k == n
            break
        end
        push!(β, nrk)
    end
    return SymTridiagonal(α, β), hcat(q...)
end
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
100-element Vector{Float64}:
 1.7763568394002505e-15
 0.17932633864614633
 0.1870482693897424
 0.19942054799790387
 0.2127314773684379
 0.22102322792303308
 0.23245088932745173
 ⋮
 5.762927109898899
 5.783619687207359
 5.79857210925601
 5.8037452659679865
 5.813869998697559
 5.823637429656785
vectors:
100×100 Matrix{Float64}:
 -0.0553607     0.0237446    -0.0245793    …  -0.027098    -0.0123004   0.048051
  0.0919687    -0.0370034     0.0381953       -0.0452025   -0.0205898   0.0807025
 -0.124413      0.0436664    -0.0447833       -0.0553489   -0.0254083   0.100331
  0.166536     -0.0482859     0.0490355       -0.0629738   -0.0292479   0.116775
 -0.219275      0.0496754    -0.0497292       -0.0718649   -0.0338922   0.137265
  0.296358     -0.0505206     0.0496206    …  -0.0785386   -0.0377234   0.155373
 -0.356127      0.0424122    -0.0404714       -0.0907859   -0.044483    0.186543
  ⋮                                        ⋱
 -2.014e-14    -0.00102184   -0.00623307      -0.0369539    0.00944734  0.00016762
  1.44031e-14   0.000861232   0.00529847   …  -0.030775     0.00776738  0.00013617
 -9.07558e-15  -0.000626429  -0.00388162      -0.0246728    0.00615499  0.000106722
  5.4486e-15    0.000427616   0.00266608      -0.0198711    0.004912    8.44311e-5
 -3.39309e-15  -0.000294114  -0.00184212      -0.0130751    0.00321372  5.49388e-5
  1.54914e-15   0.000142814   0.000896936     -0.00639179   0.00156537  2.66676e-5
```

```julia
let
    n = 1000
    graph = random_regular_graph(n, 3)
    A = laplacian_matrix(graph)
    q1 = randn(n)
    tr, Q = lanczos_reorthogonalize(A, q1; abstol=1e-5, maxiter=100)
    @info eigsolve(A, q1, 2, :SR)
    eigen(tr)
end
```

```
([3.59876e-15, 0.179326], [[-0.0316228, -0.0316228, -0.0316228, -0.0316228, -0.0316
```

Notes on Lanczos

1. In practise, we do not store all q vectors to save space.
2. Blocking technique is required if we want to compute multiple eigenvectors or a degenerate eigenvector.
3. Restarting technique can be used to improve the solution.

The Arnoldi Process

If A is not symmetric, then the orthogonal tridiagonalization $Q^T AQ = T$ does not exist in general. The Arnoldi approach involves the column by column generation of an orthogonal Q such that $Q^T AQ = H$ is a Hessenberg matrix.

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ 0 & h_{32} & h_{33} & \dots & h_{3k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & h_{kk} \end{pmatrix}$$

That is, $h_{ij} = 0$ for $i > j + 1$.

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

```
function arnoldi_iteration(A::AbstractMatrix{T}, x0::AbstractVector{T}; maxiter) where
 T
    h = Vector{T}[]
    q = [normalize(x0)]
    n = length(x0)
    @assert size(A) == (n, n)
    for k = 1:min(maxiter, n)
        u = A * q[k]    # generate next vector
        hk = zeros(T, k+1)
        for j = 1:k # subtract from new vector its components in all preceding vectors
            hk[j] = q[j]' * u
            u = u - hk[j] * q[j]
        end
        hkk = norm(u)
        hk[k+1] = hkk
        push!(h, hk)
        if abs(hkk) < 1e-8 || k >=n # stop if matrix is reducible
            break
        else
            push!(q, u ./ hkk)
        end
    end

    # construct `h`
    kmax = length(h)
    H = zeros(T, kmax, kmax)
    for k = 1:length(h)
        if k == kmax
            H[1:k, k] .= h[k][1:k]
        else
            H[1:k+1, k] .= h[k]
        end
    end
    return H, hcat(q...)
end

```

```julia
let
    n = 10
    A = randn(n, n)
    q1 = randn(n)
    h, q = arnoldi_iteration(A, q1; maxiter=100)

    # using function `KrylovKit.eigsolve`
    @info "KrylovKit.eigsolve: " eigsolve(A, q1, 2, :LR)
    # diagonalize the triangular matrix obtained with our naive implementation
    @info "Naive approach: " eigen(h).values
end;
```

```
KrylovKit.eigsolve:
eigsolve(A, q1, 2, :LR):
    ([3.63055+0.0im, 1.43317+0.456186im, 1.43317-0.456186im, 0.547984+3.09887im,

Naive approach:
(eigen(h)).values:
    [-1.24628-0.88502im, -1.24628+0.88502im, -1.17054+0.0im, -0.440888-2.87529im
```

Assignment

1. Review

I forgot to copy the definitions of `rowindices`, `colindices` and `data` in the following code. Can you help me figure out what are their possible values?

```
julia> sp = sparse(rowindices, colindices, data);

julia> sp.colptr
6-element Vector{Int64}:
 1
 2
 3
 5
 6
 6

julia> sp.rowval
5-element Vector{Int64}:
 3
 1
 1
 4
 5

julia> sp.nzval
5-element Vector{Float64}:
 0.799668435799583
 0.9421243934715178
 0.8480117750410069
 0.16419465078848616
 0.6374939310812697

julia> sp.m
5

julia> sp.n
5
```

2. Coding:

1. (easy) Implement CSC format sparse matrix-vector multiplication as function `my_spv`. Please include the following test code into your project.

```julia
using SparseArrays, Test

@testset "sparse matrix - vector multiplication" begin
    for k = 1:100
        m, n = rand(1:100, 2)
        density = rand()
        sp = sprand(m, n, density)
        v = randn(n)
        @test Matrix(sp) * v ≈ my_spv(sp, v)
    end
end
```

2. (hard) The restarting in Lanczos is a technique technique to reduce memory. Suppose we wish to calculate the p largest eigenvalues of A. If $q_1 \in \mathbb{R}^{n\times p}$ is a given normalized vector, then it can be refined as follows:

Step 1. Generate $q_2, \ldots, q_s \in \mathbb{R}^n$ via the block Lanczos algorithm.

Step 2. Form $T_s = [q_1 \mid \ldots \mid q_s]^T A [q_1 \mid \ldots \mid q_s]$, an s-by-s matrix.

Step 3. Compute an orthogonal matrix $U = [u_1 \mid \ldots \mid u_s]$ such that $U^T T_s U = \mathrm{diag}(\theta_1, \ldots, \theta_s)$ with $\theta_1 \geq \ldots \geq \theta_s$.

Step 4. Set $q_1^{(\mathrm{new})} = [q_1 \mid \ldots \mid q_s] u_1$.

Please implement a Lanczos tridiagonalization process with restarting as a Julia function. You submission should include that function as well as a test.