L4d Symmetric Eigenvalue Decomposition - Lanczos Method

Ivan Slapničar

April 9, 2018

1 Symmetric Eigenvalue Decomposition - Lanczos Method

If the matrix A is large and sparse and/or if only some eigenvalues and their eigenvectors are desired, iterative methods are the methods of choice. For example, the power method can be useful to compute the eigenvalue with the largest modulus. The basic operation in the power method is matrix-vector multiplication, and this can be performed very fast if A is sparse. Moreover, A need not be stored in the computer --- the input for the algorithm can be just a function which, given some vector x, returns the product Ax.

An *improved* version of the power method, which efficiently computes some eigenvalues (either largest in modulus or near some target value μ) and the corresponding eigenvectors, is the Lanczos method.

For more details, see [Sla14] and the references therein.

References

[Sla14] I. Slapničar, Symmetric Matrix Eigenvalue Techniques, in: L. Hogben, ed., 'Handbook of Linear Algebra', pp. 55.1-55.25, CRC Press, Boca Raton, 2014.

1.1 Prerequisites

The reader should be familiar with concepts of eigenvalues and eigenvectors, related perturbation theory, and algorithms.

1.2 Competences

The reader should be able to recognise matrices which warrant use uf Lanczos method, to apply the method and to assess the accuracy of the solution.

1.3 Lanczos method

A is a real symmetric matrix of order n.

1.3.1 Definitions

Given a nonzero vector x and an index $k < n$, the **Krylov matrix** is defined as $K_k = \begin{bmatrix} x & Ax & A^2x & \cdots & A^{k-1}x \end{bmatrix}$.

Krilov subspace is the subspace spanned by the columns of K_k.

1.3.2 Facts

1. The Lanczos method is based on the following observation. If $K_k = XR$ is the QR factorization of the matrix K_k, then the $k \times k$ matrix $T = X^T AX$ is tridiagonal. The matrices X and T can be computed by using only matrix-vector products in $O(kn)$ operations.

2. Let $T = Q\Lambda Q^T$ be the EVD of T. Then λ_i approximate well some of the largest and smallest eigenvalues of A, and the columns of the matrix $U = XQ$ approximate the corresponding eigenvectors.

3. As k increases, the largest (smallest) eigenvalues of the matrix $T_{1:k,1:k}$ converge towards some of the largest (smallest) eigenvalues of A (due to the Cauchy interlace property). The algorithm can be redesigned to compute only largest or smallest eigenvalues. Also, by using shift and invert strategy, the method can be used to compute eigenvalues near some specified value. In order to obtain better approximations, k should be greater than the number of required eigenvalues. On the other side, in order to obtain better accuracy and efficacy, k should be as small as possible.

4. The last computed element, $\mu = T_{k+1,k}$, provides information about accuracy:

$$\begin{aligned} \|AU - U\Lambda\|_2 &= \mu, \\ \|AU_{:,i} - \lambda_i U_{:,i}\|_2 &= \mu|Q_{ki}|, \quad i = 1, \ldots, k. \end{aligned}$$

   Further, there are k eigenvalues $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_k$ of A such that $|\lambda_i - \tilde{\lambda}_i| \leq \mu$, and for the corresponding eigenvectors, we have

$$\sin 2\Theta(U_{:,i}, \tilde{U}_{:,i}) \leq \frac{2\mu}{\min_{j \neq i} |\lambda_i - \tilde{\lambda}_j|}.$$

5. In practical implementations, μ is usually used to determine the index k.

6. The Lanczos method has inherent numerical instability in the floating-point arithmetic: since the Krylov vectors are, in fact, generated by the power method, they converge towards an eigenvector of A. Thus, as k increases, the Krylov vectors become more and more parallel, and the recursion in the function `myLanczos()` becomes numerically unstable and the computed columns of X cease to be sufficiently orthogonal. This affects both the convergence and the accuracy of the algorithm. For example, several eigenvalues of T may converge towards a simple eigenvalue of A (the, so called, *ghost eigenvalues*).

7. The loss of orthogonality is dealt with by using the **full reorthogonalization** procedure: in each step, the new $\mathbf{z}$ is orthogonalized against all previous columns of X, that is, in

function `myLanczos()`, the formula `z=z-Tr.dv[i]*X[:,i]-Tr.ev[i-1]*X[:,i-1]` is replaced by `z=z-sum(dot(z,Tr.dv[i])*X[:,i]-Tr.ev[i-1]*X[:,i-1]` To obtain better orthogonality, the latter formula is usually executed twice. The full reorthogonalization raises the operation count to $O(k^2n)$.

8. The **selective reorthogonalization** is the procedure in which the current z is orthogonalized against some selected columns of X, in order to attain sufficient numerical stability and not increase the operation count too much. The details are very subtle and can be found in the references.

9. The Lanczos method is usually used for sparse matrices. Sparse matrix A is stored in the sparse format in which only values and indices of nonzero elements are stored. The number of operations required to multiply some vector by A is also proportional to the number of nonzero elements.

10. The function `eigs()` implements Lanczos method real for symmetric matrices and more general Arnoldi method for general matrices.

1.3.3 Examples

```
In [1]: function myLanczos{T}(A::Array{T}, x::Vector{T}, k::Int)
            n=size(A,1)
            X=Array{T}(n,k)
            dv=Array{T}(k)
            ev=Array{T}(k-1)
            X[:,1]=x/norm(x)
            for i=1:k-1
                z=A*X[:,i]
                dv[i]=X[:,i]⋅z
                # Three-term recursion
                if i==1
                    z=z-dv[i]*X[:,i]
                else
                    # z=z-dv[i]*X[:,i]-ev[i-1]*X[:,i-1]
                    # Full reorthogonalization - once or even twice
                    z=z-sum([(z⋅X[:,j])*X[:,j] for j=1:i])
                    # z=z-sum([(z⋅X[:,j])*X[:,j] for j=1:i])
                end
                μ=norm(z)
                if μ==0
                    Tr=SymTridiagonal(dv[1:i-1],ev[1:i-2])
                    return eigvals(Tr), X[:,1:i-1]*eigvecs(Tr), X[:,1:i-1], μ
                else
                    ev[i]=μ
                    X[:,i+1]=z/μ
                end
            end
            # Last step
            z=A*X[:,end]
            dv[end]=X[:,end]⋅z
            z=z-dv[end]*X[:,end]-ev[end]*X[:,end-1]
            μ=norm(z)
            Tr=SymTridiagonal(dv,ev)
            eigvals(Tr), X*eigvecs(Tr), X, μ
        end
```

```
Out[1]: myLanczos (generic function with 1 method)
```

```
In [2]: n=100
        s=srand(421)
        A=full(Symmetric(rand(n,n)))
        # Or: A = rand(5,5) |> t -> t + t'
        x=rand(n)
        k=10
```

```
Out[2]: 10
```

```
In [3]: λ,U,X,μ=myLanczos(A,x,k)
```

```
Out[3]: ([-5.61221, -4.67728, -3.31249, -2.09514, -0.571815, 1.36414, 3.01041, 4.45828, 5.26923,
```

```
In [4]: # Orthogonality
        vecnorm(X'*X-I)
```

```
Out[4]: 2.949579672981677e-15
```

```
In [5]: X'*A*X
```

```
Out[5]: 10×10 Array{Float64,2}:
         36.5073       22.5598        3.45557e-14  …  -3.56243e-14   7.77156e-15
         22.5598       12.4312        2.7439          -2.24334e-14   4.71845e-15
          3.57214e-14   2.7439        0.520964        -2.08167e-17   1.02696e-15
         -8.573e-15    -6.59889e-15   2.76685         -2.91434e-16   1.59595e-16
         -3.09928e-14  -1.99756e-14   9.42714e-17     -1.91904e-17   1.00549e-15
          3.11903e-15   2.26902e-15   1.11022e-15  …  -9.05526e-16  -2.77556e-17
          2.95371e-14   1.8794e-14    1.31839e-16     -1.03216e-16   8.08381e-16
         -3.72792e-15  -3.83721e-15  -3.03035e-16      2.81876      -1.96024e-16
         -3.32789e-14  -1.98314e-14   3.48679e-16      0.154228      2.78294
          6.63618e-15   4.52069e-15   6.4922e-16       2.78294      -0.592886
```

```
In [6]: # Residual
        vecnorm(A*U-U*diagm(λ)), μ
```

```
Out[6]: (2.600241666735842, 2.6002416667358417)
```

```
In [7]: U'*A*U

Out[7]: 10×10 Array{Float64,2}:
         -5.61221       3.49547e-16  -1.11369e-15  ...  -4.71324e-15  -4.2414e-16
          8.06646e-16  -4.67728       7.63278e-16       2.42861e-15  -1.74652e-14
         -9.99201e-16   5.20417e-16  -3.31249           2.20657e-15  -1.45647e-14
          7.95981e-16   9.79035e-16   2.81914e-15       4.65079e-15   5.98239e-14
         -1.81143e-16   2.72352e-16   1.50964e-15       4.35589e-15   4.96283e-15
         -1.4376e-15   -5.89329e-15   7.73687e-16  ...  -1.07553e-16  -5.06925e-14
          3.38618e-15  -2.83454e-15   2.77556e-17      -7.56339e-16  -1.52586e-14
          9.62229e-16  -1.80411e-16   1.01655e-15       2.63539e-14   7.64146e-15
         -4.55452e-15   2.498e-15     2.2482e-15        5.26923      -2.77556e-16
         -1.38778e-16  -2.08167e-14  -9.4369e-15        1.11022e-16  50.0803

In [8]: # Orthogonality
        vecnorm(U'*U-I)

Out[8]: 7.60650041345998e-15

In [9]: # Full eigenvalue decomposition
        λeig,Ueig=eig(A);

In [10]: ?eigs;

search: eigs eigvecs eigvals eigvals! leading_ones leading_zeros eig eigmin



In [11]: # Lanczos method implemented in Julia
         λeigs,Ueigs=eigs(A; nev=k, which=:LM, ritzvec=true, v0=x)

Out[11]: ([50.0803, -5.70867, -5.52979, 5.36771, 5.33537, -5.17477, 5.04423, -4.91332, 4.88666,

In [12]: [λ λeigs λeig[sortperm(abs.(λeig),rev=true)[1:k]] ]

Out[12]: 10×3 Array{Float64,2}:
          -5.61221   50.0803   50.0803
          -4.67728   -5.70867  -5.70867
          -3.31249   -5.52979  -5.52979
          -2.09514    5.36771   5.36771
          -0.571815   5.33537   5.33537
           1.36414   -5.17477  -5.17477
           3.01041    5.04423   5.04423
           4.45828   -4.91332  -4.91332
           5.26923    4.88666   4.88666
          50.0803     4.6994    4.6994
```

We see that `eigs()` computes `k` eigenvalues with largest modulus. What eigenvalues did `myLanczos()` compute?

```
In [13]: for i=1:k
             println(minimum(abs,λeig-λ[i]))
         end
```

```
0.08242434872420823
0.053374157188345706
0.01005749176668047
0.0035411656264590086
0.008872819127867082
0.012090555075936482
0.10762601320997556
0.04919537417827069
0.06614007464435545
4.263256414560601e-14
```

Conslusion is that the naive implementation of Lanczos is not enough. However, it is fine, when all eigenvalues are computed:

```
In [14]: λall,Uall,Xall,μall=myLanczos(A,x,100)
```

```
Out[14]: ([-5.70867, -5.52979, -5.17477, -4.91332, -4.6239, -4.45306, -4.32179, -4.23908, -4.037
```

```
In [15]: # Residual and relative errors
         vecnorm(A*Uall-Uall*diagm(λall)), norm((λeig-λall)./λeig)
```

```
Out[15]: (8.154087610615691e-13, 9.391842171744755e-14)
```

1.3.4 Operator version

We can use Lanczos method with operator which, given vector `x`, returns the product `A*x`. We use the function `LinearMap()` from the package LinearMaps.jl

```
In [16]: # Need Pkg.add("LinearMaps"); Pkg.checkout("LinearMaps")
         using LinearMaps
```

```
In [17]: methods(LinearMap)
```

```
Out[17]: # 12 methods for generic function "(::Type)":
         (::Type{LinearMaps.LinearMap})(A::Union{AbstractArray{T,2} where T, LinearMaps.LinearMa
         (::Type{LinearMaps.LinearMap})(f, M::Int64; kwargs...) in LinearMaps at /home/slap/.jul
         (::Type{LinearMaps.LinearMap})(f, M::Int64, N::Int64; kwargs...) in LinearMaps at /home
         (::Type{LinearMaps.LinearMap})(f, M::Int64, N::Int64, T::Type; kwargs...) in LinearMaps
```

```
(::Type{LinearMaps.LinearMap})(f, fc, M::Int64; kwargs...) in LinearMaps at /home/slap/
(::Type{LinearMaps.LinearMap})(f, fc, M::Int64, N::Int64; kwargs...) in LinearMaps at /
(::Type{LinearMaps.LinearMap})(f, T::Type, args...; kwargs...) in LinearMaps at depreca
(::Type{LinearMaps.LinearMap})(f, M::Int64, T::Type; kwargs...) in LinearMaps at deprec
(::Type{LinearMaps.LinearMap})(f, fc, T::Type, args...; kwargs...) in LinearMaps at dep
(::Type{LinearMaps.LinearMap})(f, fc, M::Int64, T::Type; kwargs...) in LinearMaps at de
(::Type{LinearMaps.LinearMap})(f, fc, M::Int64, N::Int64, T::Type; kwargs...) in Linear
(::Type{T})(arg) where T in Base at sysimg.jl:77
```

```
In [18]: # Operator from the matrix
         C=LinearMap(A)
```

```
Out[18]: LinearMaps.WrappedMap{Float64,Array{Float64,2}}([0.345443 0.229319 ... 0.90542 0.642131
```

```
In [19]: λC,UC=eigs(C; nev=k, which=:LM, ritzvec=true, v0=x)
         λeigs-λC
```

```
Out[19]: 10-element Array{Float64,1}:
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
```

Here is an example of `LinearMap()` with the function.

```
In [20]: f(x)=A*x
```

```
Out[20]: f (generic function with 1 method)
```

```
In [21]: D=LinearMap(f,n,issymmetric=true)
```

```
Out[21]: LinearMaps.FunctionMap{Float64}(f, nothing, 100, 100; ismutating=false, issymmetric=tru
```

```
In [22]: λD,UD=eigs(D, nev=k, which=:LM, ritzvec=true, v0=x)
         λeigs-λD
```

```
Out[22]: 10-element Array{Float64,1}:
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
```

1.3.5 Sparse matrices

```
In [23]: ?sprand;
```

```
search: sprand sprandn StepRange StepRangeLen
```

```
In [24]: # Generate a sparse symmetric matrix
         C=sprand(n,n,0.05) |> t -> t+t'
```

```
Out[24]: 100×100 SparseMatrixCSC{Float64,Int64} with 978 stored entries:
           [5  ,    1]  =  0.804219
           [11 ,    1]  =  0.631667
           [12 ,    1]  =  0.507436
           [38 ,    1]  =  0.785969
           [40 ,    1]  =  0.972834
           [50 ,    1]  =  0.530295
           [57 ,    1]  =  0.99927
           [58 ,    1]  =  0.0550361
           [60 ,    1]  =  0.143219
           [76 ,    1]  =  0.0519795
           ⋮
           [15 ,  100]  =  0.758101
           [32 ,  100]  =  0.361693
           [57 ,  100]  =  0.258262
           [60 ,  100]  =  0.261859
           [61 ,  100]  =  0.955076
           [69 ,  100]  =  0.239199
           [71 ,  100]  =  0.728311
           [80 ,  100]  =  0.773461
           [85 ,  100]  =  0.831991
           [88 ,  100]  =  0.801726
           [90 ,  100]  =  0.745141
```

```
In [25]: issymmetric(C)
```

```
Out[25]: true
```

```
In [26]: λ,U=eigs(C; nev=k, which=:LM, ritzvec=true, v0=x)
         λ
```

```
Out[26]: 10-element Array{Float64,1}:
           5.73212
           3.68
          -3.67155
           3.5053
          -3.39346
           3.34503
          -3.19712
          -3.07929
          -3.01788
           3.00315
```