L4a Symmetric Eigenvalue Decomposition - Algorithms and Error Analysis

Ivan Slapničar

April 9, 2018

1 Symmetric Eigenvalue Decomposition - Algorithms and Error Analysis

We study only algorithms for real symmetric matrices, which are most commonly used in the applications described in this course.

For more details, see Section [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 basic linear algebra concepts and facts on eigenvalue decomposition and perturbation theory

1.2 Competences

The reader should be able to apply adequate algorithm to a given problem, and to assess accuracy of the solution.

1.3 Backward error and stability

1.3.1 Definitions

If the value of a function $f(x)$ is computed with an algorithm $\mathrm{alg(x)}$, the **algorithm error** is

$$\|\mathrm{alg(x)}-f(x)\|,$$

and the **relative algorithm error** is

$$\frac{\|\mathrm{alg}(x)-f(x)\|}{\|f(x)\|},$$

in respective norms. Therse errors can be hard or even impossible to estimate directly.

In this case, assume that $f(x)$ computed by $\mathrm{alg}(x)$ is equal to exact value of the function for a perturbed argument,

$$\mathrm{alg}(x)=f(x+\delta x),$$

for some **backward error** δx.

Algoritam is **stable** is the above equality always holds for small δx.

1.4 Basic methods

1.4.1 Definitions

The eigenvalue decomposition (EVD) of a real symmetric matrix $A=[a_{ij}]$ is $A=U\Lambda U^T$, where U is a $n\times n$ real orthonormal matrix, $U^TU=UU^T=I_n$, and $\Lambda=\mathrm{diag}(\lambda_1,\ldots,\lambda_n)$ is a real diagonal matrix.

The numbers λ_i are the eigenvalues of A, the vectors $U_{:i}$, $i=1,\ldots,n$, are the eigenvectors of A, and $AU_{:i}=\lambda_i U_{:i}$, $i=1,\ldots,n$.

If $|\lambda_1|>|\lambda_2|\geq\cdots\geq|\lambda_n|$, we say that λ_1 is the **dominant eigenvalue**.

Deflation is a process of reducing the size of the matrix whose EVD is to be determined, given that one eigenvector is known.

The **shifted matrix** of the matrix A is the matrix $A-\mu I$, where μ is the **shift**.

Power method starts from vector x_0 and computes the sequences

$$\nu_k=x_k^TAx_k,\qquad x_{k+1}=Ax_k/\|Ax_k\|,\qquad k=0,1,2,\ldots,$$

until convergence. Normalization of x_k can be performed in any norm and serves the numerical stability of the algorithm (avoiding overflow or underflow).

Inverse iteration is the power method applied to the inverse of a shifted matrix:

$$\nu_k=x_k^TAx_k,\quad x_{k+1}=(A-\mu I)^{-1}x_k,\quad x_{k+1}=x_{k+1}/\|x_{k+1}\|,\quad k=0,1,2,\ldots.$$

QR iteration starts from the matrix $A_0=A$ and forms the sequence of matrices

$$A_k=Q_kR_k\quad\text{(QR factorization)},\qquad A_{k+1}=R_kQ_k,\qquad k=0,1,2,\ldots$$

Shifted QR iteration is the QR iteration applied to a shifted matrix:

$$A_k-\mu I=Q_kR_k\quad\text{(QR factorization)},\quad A_{k+1}=R_kQ_k+\mu I,\quad k=0,1,2,\ldots$$

1.4.2 Facts

1. If λ_1 is the dominant eigenvalue and if x_0 is not orthogonal to $U_{:1}$, then $\nu_k \to \lambda_1$ and $x_k \to U_{:1}$. In other words, the power method converges to the dominant eigenvalue and its eigenvector.

2. The convergence is linear in the sense that

$$|\lambda_1 - \nu_k| \approx \left|\frac{c_2}{c_1}\right| \left|\frac{\lambda_2}{\lambda_1}\right|^k, \qquad \|U_{:1} - x_k\|_2 = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right),$$

   where c_i is the coefficient of the i-th eigenvector in the linear combination expressing the starting vector x_0.

3. Since λ_1 is not available, the convergence is determined using residuals: if $\|Ax_k - \nu_k x_k\|_2 \leq tol$, where tol is a user prescribed stopping criterion, then $|\lambda_1 - \nu_k| \leq tol$.

4. After computing the dominant eigenpair, we can perform deflation to reduce the given EVD for A to the one of size $n-1$ for A_1:

$$\begin{bmatrix} U_{:1} & X \end{bmatrix}^T A \begin{bmatrix} U_{:1} & X \end{bmatrix} = \begin{bmatrix} \lambda_1 & \\ & A_1 \end{bmatrix}, \quad \begin{bmatrix} U_{:1} & X \end{bmatrix} \text{ orthonormal}, \quad A_1 = X^T A X.$$

5. The EVD of the shifted matrix $A - \mu I$ is $U(\Lambda - \mu I)U^T$.

6. Inverse iteration requires solving the system of linear equations $(A - \mu I)x_{k+1} = x_k$ for x_{k+1} in each step. At the beginning, LU factorization of $A - \mu I$ needs to be computed, which requires $2n^3/3$ operations. In each subsequent step, two triangular systems need to be solved, which requires $2n^2$ operations.

7. If μ is close to some eigenvalue of A, the eigenvalues of the shifted matrix satisfy $|\lambda_1| \gg |\lambda_2| \geq \cdots \geq |\lambda_n|$, so the convergence of the inverse iteration method is fast.

8. If μ is very close to some eigenvalue of A, then the matrix $A - \mu I$ is nearly singular, so the solutions of linear systems may have large errors. However, these errors are almost entirely in the direction of the dominant eigenvector so the inverse iteration method is both fast and accurate.

9. We can further increase the speed of convergence of inverse iterations by substituting the shift μ with the Rayleigh quotient ν_k at the cost of computing new LU factorization.

10. Matrices A_k and A_{k+1} from both QR iterations are orthogonally similar, $A_{k+1} = Q_k^T A_k Q_k$.

11. The QR iteration method is essentially equivalent to the power method and the shifted QR iteration method is essentially equivalent to the inverse power method on the shifted matrix.

12. The straightforward application of the QR iteration requires $O(n^3)$ operations per step, so better implementation is needed.

1.4.3 Examples

In order to keep the programs simple, in the examples below we do not compute full matrix of eigenvectors.

```
In [1]: function myPower(A::Array,x::Vector,tol::Number)
            y=A*x
            ν=x⋅y
            steps=1
            while vecnorm(y-ν*x)>tol
                x=y/vecnorm(y)
                y=A*x
                ν=x⋅y
                steps+=1
            end
            ν, y/vecnorm(y), steps
        end
```

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

```
In [2]: n=6
        s=srand(421)
        A=full(Symmetric(rand(-9:9,n,n)))
```

```
Out[2]: 6×6 Array{Int64,2}:
         -8   2   1   9   2  -7
          2  -7   0  -8   1  -8
          1   0   2  -4  -3  -9
          9  -8  -4  -2   0   5
          2   1  -3   0   3   1
         -7  -8  -9   5   1  -5
```

```
In [3]: ν,x,steps=myPower(A,ones(n),1e-10)
```

```
Out[3]: (-19.992530161663694, [0.694988, -0.144771, 0.0506903, -0.528094, -0.067475, 0.458288],
```

```
In [4]: λ=eigvals(A)
```

```
Out[4]: 6-element Array{Float64,1}:
         -19.9925
         -17.0168
          -3.42589
           4.31276
           4.43503
          14.6874
```

```
In [5]: k=indmax(abs.(λ))
```

```
Out[5]: 1

In [6]: [eigvecs(A)[:,k] x]

Out[6]: 6×2 Array{Float64,2}:
          0.694988    0.694988
         -0.144771   -0.144771
          0.0506903   0.0506903
         -0.528094   -0.528094
         -0.067475   -0.067475
          0.458288    0.458288

In [7]: x

Out[7]: 6-element Array{Float64,1}:
          0.694988
         -0.144771
          0.0506903
         -0.528094
         -0.067475
          0.458288

In [8]: # Deflation
        function myDeflation(A::Array,x::Vector)
            X,R=qr(x[:,:],thin=false)
            # To make sure the returned matrix symmetric use
            # full(Symmetric(X[:,2:end]'*A*X[:,2:end]))
            X[:,2:end]'*A*X[:,2:end]
        end

Out[8]: myDeflation (generic function with 1 method)

In [9]: A₁=myDeflation(A,x)

Out[9]: 5×5 Array{Float64,2}:
         -6.57087      -0.00503334  -6.28904    1.29122    -9.41558
         -0.00503334    1.95091     -4.06933   -3.03428    -8.96407
         -6.28904      -4.06933      4.77224    1.13014    -0.624586
          1.29122      -3.03428      1.13014    3.17824     0.0515027
         -9.41558      -8.96407     -0.624586   0.0515027  -0.337992

In [10]: eigvals(A₁)

Out[10]: 5-element Array{Float64,1}:
          -17.0168
           14.6874
           -3.42589
            4.31276
            4.43503
```

```
In [11]: myPower(A₁,ones(size(A₁,1)),1e-10)

Out[11]: (-17.016810311916032, [-0.690921, -0.331625, -0.278539, 0.0114126, -0.578739], 167)

In [12]: # Put it all together - eigenvectors are ommited for the sake of simplicty
         function myPowerMethod(A::Array, tol::T) where T
             n=size(A,1)
             λ=Array{T}(n)
             for i=1:n
                 λ[i],x,steps=myPower(A,ones(n-i+1),tol)
                 A=myDeflation(A,x)
             end
             λ
         end

Out[12]: myPowerMethod (generic function with 1 method)

In [13]: myPowerMethod(A,1e-10)

Out[13]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           14.6874
            4.43503
            4.31276
           -3.42589

In [14]: # QR iteration
         function myQRIteration(A::Array, tol::Number)
             steps=1
             while norm(tril(A,-1))>tol
                 Q,R=qr(A)
                 A=R*Q
                 steps+=1
             end
             A,steps
         end

Out[14]: myQRIteration (generic function with 1 method)

In [15]: B,steps=myQRIteration(A,1e-5)
         B

Out[15]: 6×6 Array{Float64,2}:
          -19.9925          1.19793e-15   -7.85444e-17   …   3.38202e-15   8.43448e-16
           -1.01627e-20   -17.0168        -1.16907e-14       1.32061e-15  -5.19604e-16
```

```
  3.13535e-37      5.23823e-17   14.6874             1.01036e-15   8.11493e-16
  1.18738e-178   -1.7416e-159   -2.86149e-142      9.77898e-6   -2.35161e-16
  5.13609e-182    9.90527e-163   1.0578e-145       4.31276      -1.57735e-15
 -5.40376e-209   -1.1114e-189   -2.65957e-172  …  -3.25655e-27  -3.42589
```

```
In [16]: diag(B)
```

```
Out[16]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           14.6874
            4.43503
            4.31276
           -3.42589
```

1.5 Tridiagonalization

The following implementation of QR iteration requires a total of $O(n^3)$ operations:

1. Reduce A to tridiagonal form T by orthogonal similarities, $X^T AX = T$.
2. Compute the EVD of T with QR iterations, $T = Q\Lambda Q^T$.
3. Multiply $U = XQ$.

One step of QR iterations on T requires $O(n)$ operations if only Λ is computed, and $O(n^2)$ operations if Q is accumulated, as well.

1.5.1 Definitions

Given vector v, **Householder reflector** is a symmetric orthogonal matrix

$$H = I - 2\,\frac{vv^T}{v^Tv}.$$

Given $c = \cos\varphi$, $s = \sin\varphi$, and indices $i < j$, **Givens rotation matrix** is an orthogonal matrix $G(c, s, i, j)$ which is equal to the identity matrix except for elements

$$G_{ii} = G_{jj} = c, \qquad G_{ij} = -G_{ji} = s.$$

1.5.2 Facts

1. Given vector x, choosing $v = x + \operatorname{sign}(x_1)\,\|x\|_2\, e_1$ yields the Householder reflector which performs the QR factorization of x:

$$Hx = -\operatorname{sign}(a_1)\,\|a\|_2\, e_1.$$

2. Given a 2-dimensional vector $\begin{bmatrix} x \\ y \end{bmatrix}$, choosing $r = \sqrt{x^2 + y^2}$, $c = \frac{x}{r}$ and $s = \frac{y}{r}$, gives the Givens roatation matrix such that
$$G(c,s,1,2) \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}.$$
The hypotenuse r is computed using the `hypot()` function in order to avoid underflow or overflow.

3. Tridiagonal form is not unique.

4. The reduction of A to tridiagonal matrix by Householder reflections is performed as follows. Let
$$A = \begin{bmatrix} \alpha & a^T \\ a & B \end{bmatrix}.$$
Let $v = a + \operatorname{sign}(a_1) \, \|a\|_2 \, e_1$, let H be the corresponding Householder reflector and set
$$H_1 = \begin{bmatrix} 1 & \\ & H \end{bmatrix}.$$
Then
$$H_1 A H_1 = \begin{bmatrix} \alpha & a^T H \\ Ha & HBH \end{bmatrix} = \begin{bmatrix} \alpha & \nu e_1^T \\ \nu e_1 & A_1 \end{bmatrix}, \quad \nu = -\operatorname{sign}(a_1) \, \|a\|_2.$$
This step annihilates all elements in the first column below the first subdiagonal and all elements in the first row to the right of the first subdiagonal. Applying this procedure recursively yields the tridiagonal matrix $T = X^T A X$, where $X = H_1 H_2 \cdots H_{n-2}$.

5. H does not depend on the normalization of v. With the normalization $v_1 = 1$, $a_{2:n-1}$ can be overwritten by $v_{2:n-1}$, so v_1 does not need to be stored.

6. H is not formed explicitly - given v, B is overwritten with HBH in $O(n^2)$ operations by using one matrix-vector multiplication and two rank-one updates.

7. When symmetry is exploited in performing rank-2 update, tridiagonalization requires $4n^3/3$ operations. Instead of performing rank-2 update on B, one can accumulate p transformations and perform rank-$2p$ update. This **block algorithm** is rich in matrix--matrix multiplications (roughly one half of the operations is performed using BLAS 3 routines), but it requires extra workspace for U and V.

8. If X is needed explicitly, it can be computed from the stored Householder vectors v. In order to minimize the operation count, the computation starts from the smallest matrix and the size is gradually increased:
$$H_{n-2}, \quad H_{n-3}H_{n-2}, \ldots, \quad X = H_1 \cdots H_{n-2}.$$
A column-oriented version is possible as well, and the operation count in both cases is $4n^3/3$. If the Householder reflectors H_i are accumulated in the order in which they are generated, the operation count is $2n^3$.

9. The backward error bounds for functions `myTridiag()` and `myTridiagX()` are as follows: The computed matrix $\tilde{T}$ is equal to the matrix which would be obtained by exact tridiagonalization of some perturbed matrix $A + \Delta A$, where $\|\Delta A\|_2 \leq \psi \varepsilon \|A\|_2$ and ψ is a slowly increasing function of n. The computed matrix $\tilde{X}$ satisfies $\tilde{X} = X + \Delta X$, where $\|\Delta X\|_2 \leq \phi \varepsilon$ and ϕ is a slowly increasing function of n.

10. Tridiagonalization using Givens rotations requires $\frac{(n-1)(n-2)}{2}$ plane rotations, which amounts to $4n^3$ operations if symmetry is properly exploited. The operation count is reduced to $8n^3/3$ if fast rotations are used. Fast rotations are obtained by factoring out absolutely larger of c and s from G.

11. Givens rotations in the function `myTridiagG()` can be performed in different orderings. For example, the elements in the first column and row can be annihilated by rotations in the planes $(n-1,n)$, $(n-2,n-1),\ldots,(2,3)$. Givens rotations act more selectively than Householder reflectors, and are useful if A has some special structure, for example, if A is a banded matrix.

12. Error bounds for function `myTridiagG()` are the same as above, but with slightly different functions ψ and ϕ.

13. The block version of tridiagonal reduction is implemented in the LAPACK subroutine DSYTRD. The computation of X is implemented in the subroutine DORGTR. The size of the required extra workspace (in elements) is $lwork = nb * n$, where nb is the optimal block size (here, $nb = 64$), and it is determined automatically by the subroutines. The subroutine DSBTRD tridiagonalizes a symmetric band matrix by using Givens rotations. There are no Julia wappers for these routines yet!

1.5.3 Examples

```
In [17]: function myTridiag(A::Array)
             # Normalized Householder vectors are stored in the lower
             # triangular part of A below the first subdiagonal
             n=size(A,1)
             T=Float64
             A=map(T,A)
             v=Array{T}(n)
             Trid=SymTridiagonal(zeros(n),zeros(n-1))
             for j = 1 : n-2
                 μ = sign(A[j+1,j])*vecnorm(A[j+1:n, j])
                 if μ != zero(T)
                     β =A[j+1,j]+μ
                     v[j+2:n] = A[j+2:n,j] / β
                 end
                 A[j+1,j]=-μ
                 A[j,j+1]=-μ
                 v[j+1] = one(T)
                 γ = -2 / (v[j+1:n]⋅v[j+1:n])
                 w = γ* A[j+1:n, j+1:n]*v[j+1:n]
                 q = w + γ * v[j+1:n]*(v[j+1:n]⋅w) / 2
                 A[j+1:n, j+1:n] = A[j+1:n,j+1:n] + v[j+1:n]*q' + q*v[j+1:n]'
                 A[j+2:n, j] = v[j+2:n]
             end
             SymTridiagonal(diag(A),diag(A,1)), tril(A,-2)
         end
```

```
Out[17]: myTridiag (generic function with 1 method)

In [18]: A

Out[18]: 6×6 Array{Int64,2}:
          -8   2   1   9   2  -7
           2  -7   0  -8   1  -8
           1   0   2  -4  -3  -9
           9  -8  -4  -2   0   5
           2   1  -3   0   3   1
          -7  -8  -9   5   1  -5

In [19]: T,H=myTridiag(A)

Out[19]: ([-8.0 -11.7898 ... 0.0 0.0; -11.7898 -7.86331 ... 0.0 0.0; ... ; 0.0 0.0 ... -15.9075

In [20]: T

Out[20]: 6×6 SymTridiagonal{Float64}:
           -8.0     -11.7898     ⋅          ⋅          ⋅         ⋅
          -11.7898   -7.86331   3.05866     ⋅          ⋅         ⋅
            ⋅        3.05866  -1.59342  -5.10309      ⋅         ⋅
            ⋅         ⋅       -5.10309  12.128      5.23259    ⋅
            ⋅         ⋅         ⋅        5.23259  -15.9075    1.58
            ⋅         ⋅         ⋅         ⋅         1.58     4.23628

In [21]: [eigvals(A) eigvals(T)]

Out[21]: 6×2 Array{Float64,2}:
          -19.9925   -19.9925
          -17.0168   -17.0168
           -3.42589   -3.42589
            4.31276    4.31276
            4.43503    4.43503
           14.6874    14.6874

In [22]: # Extract X
         function myTridiagX(H::Array)
             n=size(H,1)
             X = eye(n)
             T=Float64
             v=Array{T}(n)
             for j = n-2 : -1 : 1
                 v[j+1] = one(T)
                 v[j+2:n] = H[j+2:n, j]
                 γ = -2 / (v[j+1:n]⋅v[j+1:n])
                 w = γ * X[j+1:n, j+1:n]'*v[j+1:n]
                 X[j+1:n, j+1:n] = X[j+1:n, j+1:n] + v[j+1:n]*w'
             end
             X
         end
```

```
Out[22]: myTridiagX (generic function with 1 method)
```

```
In [23]: X=myTridiagX(H)
```

```
Out[23]: 6×6 Array{Float64,2}:
          1.0   0.0          0.0         0.0        0.0         0.0
          0.0  -0.169638     0.34035     0.345391   0.811031    0.279854
          0.0  -0.0848189   -0.855862    0.280828   0.319165   -0.282092
          0.0  -0.76337      0.0618455  -0.536748   0.15311    -0.319219
          0.0  -0.169638    -0.380649   -0.318249  -0.0337967   0.850826
          0.0   0.593732    -0.0542644  -0.642231   0.464518   -0.127672
```

```
In [24]: # Fact 7: norm(ΔX)<ϕ*eps()
         X'*X
```

```
Out[24]: 6×6 Array{Float64,2}:
          1.0   0.0            0.0            0.0            0.0            0.0
          0.0   1.0            3.46945e-17   -5.55112e-17   5.55112e-17   -6.93889e-17
          0.0   3.46945e-17    1.0           -4.16334e-17  -6.93889e-17    5.11743e-17
          0.0  -5.55112e-17   -4.16334e-17    1.0           0.0            4.16334e-17
          0.0   5.55112e-17   -6.93889e-17    0.0           1.0           -8.32667e-17
          0.0  -6.93889e-17    5.11743e-17    4.16334e-17  -8.32667e-17    1.0
```

```
In [25]: X'*A*X
```

```
Out[25]: 6×6 Array{Float64,2}:
           -8.0          -11.7898       -1.66533e-16  …    4.44089e-16   1.11022e-15
          -11.7898        -7.86331       3.05866          -8.88178e-16   4.44089e-16
           -1.66533e-16    3.05866      -1.59342          -4.44089e-16  -1.11022e-15
            8.88178e-16    4.44089e-16  -5.10309           5.23259       7.77156e-16
            4.44089e-16    0.0          -2.22045e-16     -15.9075        1.58
            1.11022e-15    5.41234e-16  -1.10502e-15  …    1.58          4.23628
```

```
In [26]: # Tridiagonalization using Givens rotations
         function myTridiagG(A::Array)
             n=size(A,1)
             X=eye(n)
             for j = 1 : n-2
                 for i = j+2 : n
                     G,r=givens(A,j+1,i,j)
```

```
                A=(G*A)*G'
                X*=G'
            end
        end
        SymTridiagonal(diag(A),diag(A,1)), X
    end
```

```
Out[26]: myTridiagG (generic function with 1 method)
```

```
In [27]: # Lat us take a look at the `givens()` functions
         methods(givens)
```

```
Out[27]: # 3 methods for generic function "givens":
         givens(A::AbstractArray{T,2} where T, i1::Integer, i2::Integer, j::Integer) in Base.Lin
         givens(x::AbstractArray{T,1} where T, i1::Integer, i2::Integer) in Base.LinAlg at linal
         givens(f::T, g::T, i1::Integer, i2::Integer) where T in Base.LinAlg at linalg/givens.jl
```

```
In [28]: Tg,Xg=myTridiagG(map(Float64,A))
```

```
Out[28]: ([-8.0 11.7898 ... 0.0 0.0; 11.7898 -7.86331 ... 0.0 0.0; ... ; 0.0 0.0 ... -15.9075 1.
```

```
In [29]: Tg
```

```
Out[29]: 6×6 SymTridiagonal{Float64}:
          -8.0     11.7898    ⋅          ⋅          ⋅         ⋅
          11.7898  -7.86331   3.05866    ⋅          ⋅         ⋅
            ⋅       3.05866  -1.59342   5.10309     ⋅         ⋅
            ⋅        ⋅        5.10309  12.128      5.23259    ⋅
            ⋅        ⋅         ⋅        5.23259  -15.9075   1.58
            ⋅        ⋅         ⋅         ⋅         1.58     4.23628
```

```
In [30]: Xg'*Xg
```

```
Out[30]: 6×6 Array{Float64,2}:
          1.0   0.0           0.0           0.0           0.0           0.0
          0.0   1.0           1.38778e-17  -5.55112e-17   0.0           1.38778e-17
          0.0   1.38778e-17   1.0          -4.16334e-17  -5.20417e-17   7.89299e-17
          0.0  -5.55112e-17  -4.16334e-17   1.0           5.55112e-17   5.55112e-17
          0.0   0.0          -5.20417e-17   5.55112e-17   1.0           2.77556e-17
          0.0   1.38778e-17   7.89299e-17   5.55112e-17   2.77556e-17   1.0
```

```
In [31]: Xg'*A*Xg
```

```
Out[31]: 6×6 Array{Float64,2}:
          -8.0          11.7898        0.0          ...   -4.44089e-16  -4.44089e-16
          11.7898       -7.86331       3.05866             0.0           5.55112e-16
           0.0           3.05866      -1.59342             4.44089e-16  -5.55112e-16
          -8.88178e-16   4.44089e-16   5.10309             5.23259       8.88178e-16
          -4.44089e-16  -1.77636e-15   9.99201e-16       -15.9075        1.58
          -4.44089e-16   1.94289e-16  -3.55618e-16  ...    1.58          4.23628
```

1.6 Tridiagonal QR method

Let T be a real symmetric tridiagonal matrix of order n and $T = Q\Lambda Q^T$ be its EVD.

Each step of the shifted QR iterations can be elegantly implemented without explicitly computing the shifted matrix $T - \mu I$.

1.6.1 Definition

Wilkinson's shift μ is the eigenvalue of the bottom right 2×2 submatrix of T, which is closer to $T_{n,n}$.

1.6.2 Facts

1. The stable formula for the Wilkinson's shift is

$$\mu = T_{n,n} - \frac{T_{n,n-1}^2}{\tau + \operatorname{sign}(\tau)\sqrt{\tau^2 + T_{n,n-1}^2}}, \qquad \tau = \frac{T_{n-1,n-1} - T_{n,n}}{2}.$$

2. Wilkinson's shift is the most commonly used shift. With Wilkinson's shift, the algorithm always converges in the sense that $T_{n-1,n} \rightarrow 0$. The convergence is quadratic, that is, $|[T_{k+1}]_{n-1,n}| \leq c|[T_k]_{n-1,n}|^2$ for some constant c, where T_k is the matrix after the k-th sweep. Even more, the convergence is usually cubic. However, it can also happen that some $T_{i,i+i}$, $i \neq n-1$, becomes sufficiently small before $T_{n-1,n}$, so the practical program has to check for deflation at each step.

3. **Chasing the Bulge.** The plane rotation parameters at the start of the sweep are computed as if the shifted $T - \mu I$ has been formed. Since the rotation is applied to the original T and not to $T - \mu I$, this creates new nonzero elements at the positions $(3,1)$ and $(1,3)$, the so-called **bulge**. The subsequent rotations simply chase the bulge out of the lower right corner of the matrix. The rotation in the $(2,3)$ plane sets the elements $(3,1)$ and $(1,3)$ back to zero, but it generates two new nonzero elements at positions $(4,2)$ and $(2,4)$; the rotation in the $(3,4)$ plane sets the elements $(4,2)$ and $(2,4)$ back to zero, but it generates two new nonzero elements at positions $(5,3)$ and $(3,5)$, etc.

4. The effect of this procedure is the following. At the end of the first sweep, the resulting matrix T_1 is equal to the the matrix that would have been obtained by factorizing $T - \mu I = QR$ and computing $T_1 = RQ + \mu I$.

5. Since the convergence of the function `myTridEigQR()` is quadratic (or even cubic), an eigenvalue is isolated after just a few steps, which requires $O(n)$ operations. This means that $O(n^2)$ operations are needed to compute all eigenvalues.

6. If the eigenvector matrix Q is desired, the plane rotations need to be accumulated similarly to the accumulation of X in the function `myTridiagG()`. This accumulation requires $O(n^3)$ operations. Another, faster, algorithm to first compute only Λ and then compute Q using inverse iterations. Inverse iterations on a tridiagonal matrix are implemented in the LAPACK routine DSTEIN.

7. **Error bounds.** Let $U\Lambda U^T$ and $\tilde{U}\tilde{\Lambda}\tilde{U}^T$ be the exact and the computed EVDs of A, respectively, such that the diagonals of Λ and $\tilde{\Lambda}$ are in the same order. Numerical methods generally compute the EVD with the errors bounded by

$$|\lambda_i - \tilde{\lambda}_i| \leq \phi\epsilon \|A\|_2, \qquad \|u_i - \tilde{u}_i\|_2 \leq \psi\epsilon \frac{\|A\|_2}{\min_{j\neq i}|\lambda_i - \tilde{\lambda}_j|},$$

where ϵ is machine precision and ϕ and ψ are slowly growing polynomial functions of n which depend upon the algorithm used (typically $O(n)$ or $O(n^2)$). Such bounds are obtained by combining perturbation bounds with the floating-point error analysis of the respective algorithms.

8. The eigenvalue decomposition $T = Q\Lambda Q^T$ computed by `myTridEigQR()` satisfies the error bounds from fact 7. with A replaced by T and U replaced by Q. The deflation criterion implies $|T_{i,i+1}| \leq \epsilon \|T\|_F$, which is within these bounds.

9. The EVD computed by function `mySymEigQR()` satisfies the error bounds given in Fact 7. However, the algorithm tends to perform better on matrices, which are graded downwards, that is, on matrices that exhibit systematic decrease in the size of the matrix elements as we move along the diagonal.
For such matrices the tiny eigenvalues can usually be computed with higher relative accuracy (although counterexamples can be easily constructed). If the tiny eigenvalues are of interest, it should be checked whether there exists a symmetric permutation that moves larger elements to the upper left corner, thus converting the given matrix to the one that is graded downwards.

10. The function `myTridEigQR()` is implemented in the LAPACK subroutine DSTEQR. This routine can compute just the eigenvalues, or both eigenvalues and eigenvectors.

11. The function `mySymEigQR()` is Algorithm 5 is implemented in the functions `eig()`, `eigvals()` and `eigvecs()`, and in the LAPACK routine DSYEV. To compute only eigenvalues, DSYEV calls DSYTRD and DSTEQR without the eigenvector option. To compute both eigenvalues and eigenvectors, DSYEV calls DSYTRD, DORGTR, and DSTEQR with the eigenvector option.

1.6.3 Examples

```
In [32]: function myTridEigQR(A1::SymTridiagonal)
             A=deepcopy(A1)
             n=length(A.dv)
             T=Float64
             λ=Array{T}(n)
             B=Array{T}
             if n==1
                 return map(T,A.dv)
             end
             if n==2
                 τ=(A.dv[end-1]-A.dv[end])/2
                 μ=A.dv[end]-A.ev[end]^2/(τ+sign(τ)*sqrt(τ^2+A.ev[end]^2))
```

```
            # Only rotation
            B=A[1:2,1:2]
            G,r=givens(B-μ*I,1,2,1)
            B=(G*B)*G'
            return diag(B)[1:2]
        end
        steps=1
        k=0
        while k==0 && steps<=10
            # Shift
            τ=(A.dv[end-1]-A.dv[end])/2
            μ=A.dv[end]-A.ev[end]^2/(τ+sign(τ)*sqrt(τ^2+A.ev[end]^2))
            # First rotation
            B=A[1:3,1:3]
            G,r=givens(B-μ*I,1,2,1)
            B=(G*B)*G'
            A.dv[1:2]=diag(B)[1:2]
            A.ev[1:2]=diag(B,-1)
            bulge=B[3,1]
            # Bulge chasing
            for i = 2 : n-2
                B=A[i-1:i+2,i-1:i+2]
                B[3,1]=bulge
                B[1,3]=bulge
                G,r=givens(B,2,3,1)
                B=(G*B)*G'
                A.dv[i:i+1]=diag(B)[2:3]
                A.ev[i-1:i+1]=diag(B,-1)
                bulge=B[4,2]
            end
            # Last rotation
            B=A[n-2:n,n-2:n]
            B[3,1]=bulge
            B[1,3]=bulge
            G,r=givens(B,2,3,1)
            B=(G*B)*G'
            A.dv[n-1:n]=diag(B)[2:3]
            A.ev[n-2:n-1]=diag(B,-1)
            steps+=1
            # Deflation criterion
            k=findfirst(abs.(A.ev) .< sqrt.(abs.(A.dv[1:n-1].*A.dv[2:n]))*eps(T))
        end
        λ[1:k]=myTridEigQR(SymTridiagonal(A.dv[1:k],A.ev[1:k-1]))
        λ[k+1:n]=myTridEigQR(SymTridiagonal(A.dv[k+1:n],A.ev[k+1:n-1]))
        λ
    end
```

```
Out[32]: myTridEigQR (generic function with 1 method)
```

```
In [33]: λ=eigvals(T)
```

```
Out[33]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           -3.42589
            4.31276
            4.43503
           14.6874
```

```
In [34]: λ1=myTridEigQR(T)
```

```
Out[34]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           14.6874
           -3.42589
            4.43503
            4.31276
```

```
In [35]: (sort(λ)-sort(λ1))./sort(λ)
```

```
Out[35]: 6-element Array{Float64,1}:
          -5.33106e-16
          -6.2633e-16
          -1.16665e-15
           0.0
           0.0
          -1.20944e-16
```

1.6.4 Computing the eigenvectors

Once the eigenvalues are computed, the eigeenvectors can be efficiently computed with inverse iterations. Inverse iterations for tridiagonal matrices are implemented in the LAPACK routine DSTEIN.

```
In [36]: U=LAPACK.stein!(T.dv,T.ev,λ)
```

```
Out[36]: 6×6 Array{Float64,2}:
           0.694988    -0.03569     0.276879    0.353131   -0.5599      0.0294336
           0.706937    -0.0272955  -0.107421   -0.368794    0.590541   -0.0566397
          -0.124501    -0.0558835   0.911406   -0.106945    0.216287   -0.304136
          -0.0251675   -0.18526     0.262891   -0.0972702   0.0984485   0.936365
           0.0330718    0.977373    0.107407    0.0409805   0.0661956   0.161405
          -0.00215667  -0.0726602  -0.0221482   0.846588    0.526231    0.0244011
```

```
In [37]: # Orthogonality
         vecnorm(U'*U-I)
```

```
Out[37]: 5.205466701590827e-16
```

```
In [38]: # Residual
         vecnorm(T*U-U*diagm(λ))
```

```
Out[38]: 1.1258410421055917e-14
```

```
In [39]: # Some timings - n=100, 200, 400
         n=400
         Tbig=SymTridiagonal(rand(n),rand(n-1))
         @time myTridEigQR(Tbig);
         @time λbig=eigvals(Tbig);
         @time LAPACK.stein!(Tbig.dv,Tbig.ev,λbig);
```

```
  0.327687 seconds (2.29 M allocations: 219.497 MiB, 27.99% gc time)
  0.006061 seconds (12 allocations: 13.234 KiB)
  0.021522 seconds (24 allocations: 1.266 MiB)
```

```
In [40]: n=2000
         Tbig=SymTridiagonal(rand(n),rand(n-1))
         @time λbig=eigvals(Tbig);
         @time U=LAPACK.stein!(Tbig.dv,Tbig.ev,λbig);
         @time eig(Tbig);
```

```
  0.158401 seconds (12 allocations: 63.141 KiB)
  0.587675 seconds (26 allocations: 30.722 MiB, 1.62% gc time)
  0.959685 seconds (72.08 k allocations: 35.003 MiB, 4.57% gc time)
```

Alternatively, the rotations in `myTridEigQR()` can be accumulated to compute the eigenvctors. This is not optimal, but is instructive. We keep the name of the function, using Julia's **multiple dispatch** feature.

```
In [41]: function myTridEigQR(A1::SymTridiagonal,U::Array)
             # U is either the identity matrix or the output from myTridiagX()
             A=deepcopy(A1)
             n=length(A.dv)
             T=Float64
             λ=Array{T}(n)
             B=Array{T}
             if n==1
                 return map(T,A.dv), U
```

```
end
if n==2
    τ=(A.dv[end-1]-A.dv[end])/2
    μ=A.dv[end]-A.ev[end]^2/(τ+sign(τ)*sqrt(τ^2+A.ev[end]^2))
    # Only rotation
    B=A[1:2,1:2]
    G,r=givens(B-μ*I,1,2,1)
    B=(G*B)*G'
    U*=G'
    return diag(B)[1:2], U
end
steps=1
k=0
while k==0 && steps<=10
    # Shift
    τ=(A.dv[end-1]-A.dv[end])/2
    μ=A.dv[end]-A.ev[end]^2/(τ+sign(τ)*sqrt(τ^2+A.ev[end]^2))
    # First rotation
    B=A[1:3,1:3]
    G,r=givens(B-μ*I,1,2,1)
    B=(G*B)*G'
    U[:,1:3]*=G'
    A.dv[1:2]=diag(B)[1:2]
    A.ev[1:2]=diag(B,-1)
    bulge=B[3,1]
    # Bulge chasing
    for i = 2 : n-2
        B=A[i-1:i+2,i-1:i+2]
        B[3,1]=bulge
        B[1,3]=bulge
        G,r=givens(B,2,3,1)
        B=(G*B)*G'
        U[:,i-1:i+2]=U[:,i-1:i+2]*G'
        A.dv[i:i+1]=diag(B)[2:3]
        A.ev[i-1:i+1]=diag(B,-1)
        bulge=B[4,2]
    end
    # Last rotation
    B=A[n-2:n,n-2:n]
    B[3,1]=bulge
    B[1,3]=bulge
    G,r=givens(B,2,3,1)
    B=(G*B)*G'
    U[:,n-2:n]*=G'
    A.dv[n-1:n]=diag(B)[2:3]
    A.ev[n-2:n-1]=diag(B,-1)
    steps+=1
    # Deflation criterion
```

```
            k=findfirst(abs.(A.ev) .< sqrt.(abs.(A.dv[1:n-1].*A.dv[2:n]))*eps())
        end
        λ[1:k], U[:,1:k]=myTridEigQR(SymTridiagonal(A.dv[1:k],A.ev[1:k-1]),U[:,1:k])
        λ[k+1:n], U[:,k+1:n]=myTridEigQR(SymTridiagonal(A.dv[k+1:n],A.ev[k+1:n-1]),U[:,k+1:
        λ, U
    end
```

```
Out[41]: myTridEigQR (generic function with 2 methods)
```

```
In [43]: λ,U=myTridEigQR(T,eye(T))
         λ
```

```
Out[43]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           14.6874
           -3.42589
            4.43503
            4.31276
```

```
In [44]: U
```

```
Out[44]: 6×6 Array{Float64,2}:
           0.694988     0.03569    -0.0294336  -0.276879    0.5599      0.353131
           0.706937     0.0272955   0.0566397   0.107421   -0.590541   -0.368794
          -0.124501     0.0558835   0.304136   -0.911406   -0.216287   -0.106945
          -0.0251675    0.18526    -0.936365   -0.262891   -0.0984485  -0.0972702
           0.0330718   -0.977373   -0.161405   -0.107407   -0.0661956   0.0409805
          -0.00215667   0.0726602  -0.0244011   0.0221482  -0.526231    0.846588
```

```
In [45]: # Orthogonality
         vecnorm(U'*U-I)
```

```
Out[45]: 1.0862687023937815e-15
```

```
In [46]: # Residual
         vecnorm(T*U-U*diagm(λ))
```

```
Out[46]: 1.7658748820255334e-14
```

1.7 Symmetric QR method

Combining `myTridiag()`, `myTridiagX()` and `myTridEigQR()`, we get the method for computing symmetric EVD.

```
In [47]: function mySymEigQR(A::Array)
             T,H=myTridiag(A)
             X=myTridiagX(H)
             myTridEigQR(T,X)
         end

Out[47]: mySymEigQR (generic function with 1 method)

In [48]: λ,U=mySymEigQR(map(Float64,A))
         λ

Out[48]: 6-element Array{Float64,1}:
          -19.9925
          -17.0168
           14.6874
           -3.42589
            4.43503
            4.31276

In [49]: U

Out[49]: 6×6 Array{Float64,2}:
           0.694988    0.03569     -0.0294336  -0.276879    0.5599      0.353131
          -0.144771   -0.693969    -0.36724    -0.500131   -0.208393    0.262724
           0.0506903  -0.330558    -0.572692    0.65657     0.334872   -0.130241
          -0.528094   -0.289658     0.461241   -0.0207771   0.648115    0.0631494
          -0.067475    0.00999187   0.157314    0.434842   -0.231655    0.85314
           0.458288   -0.56909      0.546628    0.229353   -0.239224   -0.239741

In [50]: # Orthogonality
         norm(U'*U-I)

Out[50]: 1.1805055554232733e-15

In [51]: # Residual
         vecnorm(A*U-U*diagm(λ))

Out[51]: 2.988517724662741e-14
```

1.8 Unsymmetric matrices

The QR iterations for unsymmetric matrices are implemented as follows:

1. Reduce A to Hessenberg form form H by orthogonal similarities, $X^T AX = H$.
2. Compute the EVD of H with QR iterations, $H = Q\Lambda Q^*$.

3. Multiply $U = XQ$.

The algorithm requires of $O(n^3)$ operations. For more details, see Section **??** and the references therein.

```
In [52]: A=rand(-9:9,5,5)
```

```
Out[52]: 5×5 Array{Int64,2}:
          -1   4   6  -2  -9
          -7   6  -8  -5  -5
          -8   3   0  -7  -9
           0   7  -9   2   3
           0  -6   3  -4   4
```

```
In [53]: λ,X=eig(A)
         λ
```

```
Out[53]: 5-element Array{Complex{Float64},1}:
             4.69648+6.89506im
             4.69648-6.89506im
           -0.255854+5.41527im
           -0.255854-5.41527im
             2.11875+0.0im
```

```
In [54]: X
```

```
Out[54]: 5×5 Array{Complex{Float64},2}:
            0.634239+0.0im        0.634239-0.0im       ...  -0.75577+0.0im
          -0.175509+0.338943im  -0.175509-0.338943im      0.185923+0.0im
          0.0578582+0.166185im  0.0578582-0.166185im      0.311765+0.0im
          -0.151699+0.371527im  -0.151699-0.371527im      0.194408+0.0im
          -0.407157-0.307031im  -0.407157+0.307031im      0.509169+0.0im
```

```
In [55]: # Residual
         vecnorm(A*X-X*diagm(λ))
```

```
Out[55]: 1.652354479632358e-14
```

1.8.1 Hessenberg factorization

```
In [56]: @which hessfact(A)
```

```
Out[56]: hessfact(A::Union{Base.ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Base.Multiplicati
```

```
In [57]: B=hessfact(A)
```

```
Out[57]: Base.LinAlg.Hessenberg{Float64,Array{Float64,2}}([-1.0 -7.14948 ... -5.96295 -4.33787;

In [58]: B[:H]

Out[58]: 5×5 Array{Float64,2}:
          -1.0     -7.14948    -5.61347  -5.96295  -4.33787
          10.6301   0.123894    3.10093   3.0168   14.9054
           0.0      3.96474     7.55157   3.79067  -3.16602
           0.0      0.0       -10.4986    3.93755   6.33142
           0.0      0.0         0.0       2.23559   0.386986

In [59]: B[:Q]

Out[59]: 5×5 Base.LinAlg.HessenbergQ{Float64,Array{Float64,2}}:
          1.0   0.0        0.0        0.0          0.0
          0.0  -0.658505   0.542575  -0.0127722  -0.521364
          0.0  -0.752577  -0.474754   0.0111756   0.456194
          0.0   0.0        0.545725  -0.602219    0.58268
          0.0   0.0        0.427089   0.798151    0.424912

In [60]: B[:Q]'*A*B[:Q]

Out[60]: 5×5 Array{Float64,2}:
          -1.0          -7.14948       -5.61347      -5.96295  -4.33787
          10.6301        0.123894       3.10093       3.0168   14.9054
           4.44089e-16   3.96474        7.55157       3.79067  -3.16602
          -1.38778e-17  -1.77636e-15  -10.4986        3.93755   6.33142
           0.0           2.77556e-17    6.10623e-16   2.23559   0.386986

In [61]: eigvals(B[:H])

Out[61]: 5-element Array{Complex{Float64},1}:
            4.69648+6.89506im
            4.69648-6.89506im
          -0.255854+5.41527im
          -0.255854-5.41527im
            2.11875+0.0im
```