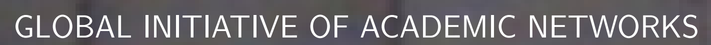


Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module B - Eigenvalue and Singular Value Decompositions

IIT INDORE, 2016

GLOBAL INITIATIVE OF ACADEMIC NETWORKS

Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module B - Eigenvalue and Singular Value Decompositions

https://github.com/ivanslapnicar/GIAN-Applied-NLA-Course

Cover photo: TU Berlin, Institut für Mathematik

IIT INDORE, 2016

Contents

1 Eigenvalue Decomposition - Definitions and Facts **5**

1.1 Prerequisites . . . 5
1.2 Competences . . . 5
1.3 Selected references . . . 5
1.4 General matrices . . . 5
1.4.1 Definitions . . . 5
1.4.2 Facts . . . 6
1.4.3 Examples . . . 7
1.4.4 Example . . . 9
1.4.5 Example . . . 10
1.4.6 Example - Circulant matrix . . . 12
1.5 Hermitian and real symmetric matrices . . . 14
1.5.1 Definitions . . . 14
1.5.2 Facts . . . 14
1.5.3 Example - Hermitian matrix . . . 16
1.5.4 Example - real symmetric matrix . . . 21
1.6 Positive definite matrices . . . 22
1.6.1 Definitions . . . 22
1.6.2 Facts . . . 22
1.6.3 Example - Positive definite matrix . . . 23
1.6.4 Example - Positive semidefinite matrix . . . 24
1.6.5 Example - Covariance and corellation matrices . . . 25

2 Eigenvalue Decomposition - Perturbation Theory **28**

2.1 Prerequisites . . . 28
2.2 Competences . . . 28
2.3 Norms . . . 28
2.3.1 Definitions . . . 28
2.3.2 Examples . . . 29
2.3.3 Facts . . . 29
2.4 Errors and condition numbers . . . 32
2.4.1 Definitions . . . 32
2.5 Peturbation bounds . . . 32
2.5.1 Definitions . . . 32
2.5.2 Facts . . . 33
2.5.3 Example - Nondiagonalizable matrix . . . 34
2.5.4 Example - Jordan form . . . 36
2.5.5 Example - Normal matrix . . . 40
2.5.6 Example - Hermitian matrix . . . 41

3 Symmetric Eigenvalue Decomposition - Algorithms and Error Analysis **47**

3.1 Prerequisites . . . 47
3.2 Competences . . . 47
3.3 Backward error and stability . . . 47

3.3.1 Definitions . . . 47
3.4 Basic methods . . . 47
3.4.1 Definitions . . . 47
3.4.2 Facts . . . 48
3.4.3 Examples . . . 49
3.5 Tridiagonalization . . . 52
3.5.1 Facts . . . 52
3.6 Tridiagonal QR method . . . 57
3.6.1 Definition . . . 57
3.6.2 Facts . . . 57
3.6.3 Examples . . . 58
3.6.4 Computing the eigenvectors . . . 60
3.6.5 Symmetric QR method . . . 63

4 Symmetric Eigenvalue Decomposition - Algorithms for Tridiagonal Matrices 65
4.1 Prerequisites . . . 65
4.2 Competences . . . 65
4.3 Bisection and inverse iteration . . . 65
4.3.1 Facts . . . 65
4.4 Divide-and-conquer . . . 67
4.4.1 Facts . . . 67
4.5 MRRR . . . 70

5 Symmetric Eigenvalue Decomposition - Jacobi Method and High Relative Accuracy 72
5.1 Prerequisites . . . 72
5.2 Competences . . . 72
5.3 Jacobi method . . . 72
5.3.1 Definitions . . . 72
5.3.2 Facts . . . 73
5.3.3 Examples . . . 73
5.4 Relative perturbation theory . . . 76
5.4.1 Definition . . . 76
5.4.2 Facts . . . 76
5.4.3 Example - Scaled matrix . . . 77
5.5 Indefinite matrices . . . 79
5.5.1 Definition . . . 79
5.5.2 Facts . . . 79
5.6 # Symmetric Eigenvalue Decomposition - Lanczos Method . . . 79
5.7 Prerequisites . . . 80
5.8 Competences . . . 80
5.9 Lanczos method . . . 80
5.9.1 Definitions . . . 80
5.9.2 Facts . . . 80
5.9.3 Examples . . . 81

5.9.4 Operator version . . . 90
5.9.5 Sparse matrices . . . 93

6 Singular Value Decomposition - Definitions and Facts **95**
6.1 Prerequisites . . . 95
6.2 Competences . . . 95
6.3 Selected references . . . 95
6.4 Singular value problems . . . 95
6.4.1 Definitions . . . 95
6.4.2 Facts . . . 95
6.4.3 Example - Symbolic computation . . . 97
6.4.4 Example - Random complex matrix . . . 98
6.4.5 Example - Random real matrix . . . 100
6.5 Matrix approximation . . . 102

7 Singular Value Decomposition - Perturbation Theory **104**
7.1 Prerequisites . . . 104
7.2 Competences . . . 104
7.3 Peturbation bounds . . . 104
7.3.1 Definitions . . . 104
7.3.2 Facts . . . 104
7.3.3 Example . . . 105
7.4 Relative perturbation theory . . . 108
7.4.1 Definitions . . . 108
7.4.2 Facts . . . 108
7.4.3 Example - Bidiagonal matrix . . . 109

8 Singular Value Decomposition - Algorithms and Error Analysis **113**
8.1 Prerequisites . . . 113
8.2 Competences . . . 113
8.3 Basics . . . 113
8.3.1 Definitions . . . 113
8.3.2 Facts . . . 113
8.4 Bidiagonalization . . . 114
8.4.1 Facts . . . 114
8.5 Bidiagonal QR method . . . 118
8.5.1 Facts . . . 118
8.5.2 Examples . . . 119
8.6 QR method . . . 122

9 Singular Value Decomposition - Jacobi and Lanczos Methods **125**
9.1 Prerequisites . . . 125
9.2 Competences . . . 125
9.3 One-sided Jacobi method . . . 125
9.3.1 Definition . . . 125
9.3.2 Facts . . . 125

9.3.3 Example - Standard matrix . . . 126
9.3.4 Example - Strongly scaled matrix . . . 128
9.4 Lanczos method . . . 132
9.4.1 Example - Large matrix . . . 135
9.4.2 Example - Very large sparse matrix . . . 135

10 Algorithms for Structured Matrices 138
10.1 Prerequisites . . . 138
10.2 Competences . . . 138
10.3 Rank revealing decompositions . . . 138
10.3.1 Definitions . . . 138
10.3.2 Facts . . . 138
10.3.3 Example - Positive definite matrix . . . 140
10.3.4 Example - Hilbert matrix . . . 142
10.4 Symmetric arrowhead and DPR1 matrices . . . 149
10.4.1 Definitions . . . 149
10.4.2 Facts on arrowhead matrices . . . 149
10.4.3 Example - Random arrowhead matrix . . . 150
10.4.4 Example - Numerically demanding matrix . . . 152
10.4.5 Facts on DPR1 matrices . . . 152
10.4.6 Example - Random DPR1 matrix . . . 153
10.4.7 Example - Numerically demanding matrix . . . 153

11 Updating the SVD 155
11.1 Prerequisites . . . 155
11.2 Competences . . . 155
11.3 Facts . . . 155
11.3.1 Example - Adding row to a tall matrix . . . 156
11.3.2 Example - Adding row to a flat matrix . . . 158
11.3.3 Example - Adding columns . . . 158
11.3.4 Example - Updating a low rank approximation . . . 159

12 Tutorial 2 - Examples in Eigenvalue Decomposition 161
12.1 Assignment 1 . . . 161
12.2 Assignment 2 . . . 161
12.3 Assignment 3* . . . 161

13 Tutorial 3 - Examples in Singular Value Decomposition 162
13.1 Assignment 1 . . . 162
13.2 Assignment 2 . . . 162
13.3 Assignment 3 . . . 162

14 Tutorial 4 - Examples in SVD Updating 163
14.1 Assignment 1* . . . 163
14.2 Assignment 2** . . . 163

1 Eigenvalue Decomposition - Definitions and Facts

1.1 Prerequisites

The reader should be familiar with basic linear algebra concepts.

1.2 Competences

The reader should be able to undestand and check the facts about eigenvalue decomposition.

1.3 Selected references

There are many excellent books on the subject. Here we list a few:
J. W. Demmel, Applied Numerical Linear Algebra
G. H. Golub and C. F. Van Loan, Matrix Computations
N. Higham, Accuracy and Stability of Numerical Algorithms
L. Hogben, ed., Handbook of Linear Algebra
B. N. Parlett, The Symmetric Eigenvalue Problem
G. W. Stewart, Matrix Algorithms, Vol. II: Eigensystems
L. N. Trefethen and D. Bau, III, Numerical Linear Algebra
J. H. Wilkinson, The Algebraic Eigenvalue Problem

1.4 General matrices

For more details and the proofs of the Facts below, see L. M. DeAlba, Determinants and Eigenvalues and the references therein.

1.4.1 Definitions

We state the basic definitions:
Let $F = \mathbb{R}$ or $F = \mathbb{C}$ and let $A \in F^{n\times n}$ with elements $a_{ij} \in F$.
An element $\lambda \in F$ is an **eigenvalue** of A if $\exists x \in F$, $x \neq 0$ such that

$$Ax = \lambda x,$$

and x is an **eigenvector** of λ.
Characteristic polynomial of A is $p_A(x) = \det(A - xI)$.
Algebraic multiplicty, $\alpha(\lambda)$, is the multiplicity of λ as a root of $p_A(x)$.
Spectrum of A, $\sigma(A)$, is the multiset of all eigenvalues of A, with each eigenvalue appearing $\alpha(A)$ times.
Spectral radius of A is

$$\rho(A) = \max\{|\lambda|, \lambda \in \sigma(A)\}.$$

Eigenspace of λ is

$$E_\lambda(A) = \ker(A - \lambda I).$$

Geometric multiplicity of λ is

$$\gamma(\lambda) = \dim(E_\lambda(A)).$$

λ is **simple** if $\alpha(\lambda) = 1$.
λ is **semisimple** if $\alpha(\lambda) = \gamma(\lambda)$.
A is **nonderogatory** if $\gamma(\lambda) = 1$ for all λ.
A is **nondefective** if every λ is semisimple.
A is **diagonalizable** if there exists nonsingular B matrix and diagonal matrix D such that $A = BDB^{-1}$.
Trace of A is

$$\mathrm{tr}(A) = \sum_i a_{ii}.$$

$Q \in \mathbb{C}^{n\times n}$ is **unitary** if $Q^*Q = QQ^* = I$, where $Q^* = (\bar{Q})^T$.
Schur decomposition of A is $A = QTQ^*$, where Q is unitary and T is upper triangular.
A and B are **similar** if $B = QAQ^{-1}$ for some nonsingular matrix Q.
A is **normal** if $AA^* = A^*A$.

1.4.2 Facts

There are many facts related to the eigenvalue problem for general matrices. We state some basic ones:

1. $\lambda \in \sigma(A) \Leftrightarrow p_A(\lambda) = 0$.
2. $p_A(A) = 0$. *(Cayley-Hamilton Theorem)*
3. A simple eigenvalue is semisimple.
4. $\mathrm{tr}(A) = \sum_{i=1}^n \lambda_i$.
5. $\det(A) = \prod_{i=1}^n \lambda_i$.
6. A is singular $\Leftrightarrow \det(A) = 0 \Leftrightarrow 0 \in \sigma(A)$.
7. If A is triangular, $\sigma(A) = \{a_{11}, a_{22}, \dots, a_{nn}\}$.
8. For $A \in \mathbb{C}^{n\times n}$, $\lambda \in \sigma(A) \Leftrightarrow \bar{\lambda} \in \sigma(A^*)$.
9. For $A \in \mathbb{R}^{n\times n}$, $\lambda \in \sigma(A) \Leftrightarrow \bar{\lambda} \in \sigma(A)$. *(Corollary of the Fundamental theorem of algebra)*
10. If (λ, x) is an eigenpair of a nonsingular A, then $(1/\lambda, x)$ is an eigenpair of A^{-1}.
11. Eigenvectors corresponding to distinct eigenvalues are linearly independent.
12. A is diagonalizable $\Leftrightarrow$ A is nondefective $\Leftrightarrow$ A has n linearly independent eigenvectors.
13. Every A has Schur decomposition. Moreover, $T_{ii} = \lambda_i$.
14. If A is normal, matrix T from its Schur decomposition is normal. Consequently:
    - T is diagonal, and has eigenvalues of A on diagonal,
    - matrix Q of the Schur decomposition is the unitary matrix of eigenvectors,
    - all eigenvalues of A are semisimple and A is nondefective.

15. If A and B are similar, $\sigma(A) = \sigma(B)$. Consequently, $\operatorname{tr}(A) = \operatorname{tr}(B)$ and $\det(A) = \det(B)$.

16. Eigenvalues and eigenvectors are continous and differentiable: if λ is a simple eigenvalue of A and $A(\varepsilon) = A + \varepsilon E$ for some $E \in F^{n\times n}$, for small ε there exist differentiable functions $\lambda(\varepsilon)$ and $x(\varepsilon)$ such that
$$A(\varepsilon)x(\varepsilon) = \lambda(\varepsilon)x(\varepsilon).$$

17. Classical motivation for the eigenvalue problem is the following: consider the system of linear differential equations with constant coefficients,
$$\dot{y}(t) = Ay(t).$$
If the solution is $y = e^{\lambda t}x$ for some constant vector x, then $\lambda e^{\lambda t}x = Ae^{\lambda t}x$, or $Ax = \lambda x$.

1.4.3 Examples

We shall illustrate above Definitions and Facts on several small examples, using symbolic computation:

```
In [1]: using SymPy
```

```
In [2]: A=[-3 7 -1; 6 8 -2; 72 -28 19]
```

```
Out[2]: 3x3 Array{Int64,2}:
         -3    7  -1
          6    8  -2
         72  -28  19
```

```
In [3]: @vars x
```

```
Out[3]: (x,)
```

```
In [4]: A-x*I
```

Out[4]:

$$\begin{bmatrix} -x-3 & 7 & -1 \\ 6 & -x+8 & -2 \\ 72 & -28 & -x+19 \end{bmatrix}$$

```
In [5]: # Characteristic polynomial p_A(λ)
        p(x)=det(A-x*I)
        p(x)
```

Out[5]:

$$(-x-3)\left(-x+8-\frac{42}{-x-3}\right)\left(-x-\frac{\left(-28-\frac{504}{-x-3}\right)\left(-2+\frac{6}{-x-3}\right)}{-x+8-\frac{42}{-x-3}}+19+\frac{72}{-x-3}\right)$$

```
In [6]: # Characteristic polynomial in nicer form
        p(x)=factor(det(A-x*I))
        p(x)
```

Out[6]:

$$-(x-15)^2(x+6)$$

```
In [7]: λ=solve(p(x),x)
```

Out[7]:

$$\begin{bmatrix}-6\\15\end{bmatrix}$$

The eigenvalues are $\lambda_1 = -6$ and $\lambda_2 = 15$ with algebraic multiplicities $\alpha(\lambda_1) = 1$ and $\alpha(\lambda_2) = 2$.

```
In [8]: g=nullspace(A-λ[1]*I)

Out[8]: 1-element Array{Any,1}:
         SymPy.SymMatrix(PyObject Matrix([
        [-1/4],
        [ 1/4],
        [   1]]))

In [9]: h=nullspace(A-λ[2]*I)

Out[9]: 1-element Array{Any,1}:
         SymPy.SymMatrix(PyObject Matrix([
        [-1/4],
        [-1/2],
        [   1]]))
```

The geometric multiplicities are $\gamma(\lambda_1) = 1$ and $\gamma(\lambda_2) = 1$. Thus, λ_2 is not semisimple, therefore A is defective and not diagonalizable.

```
In [10]: # Traace and determinant
         trace(A), λ[1]+λ[2]+λ[2]

Out[10]: (24,24)

In [11]: det(A), λ[1]*λ[2]*λ[2]

Out[11]: (-1350.0000000000002,-1350)

In [12]: # Schur decomposition
         T,Q=schur(A)

Out[12]: (
         3x3 Array{Float64,2}:
          -6.0  25.4662        -72.2009
           0.0  15.0           -12.0208
           0.0   1.48587e-15    15.0    ,

         3x3 Array{Float64,2}:
          -0.235702  -0.0571662  -0.970143
           0.235702  -0.971825   -5.90663e-16
           0.942809   0.228665   -0.242536    ,

         Complex{Float64}[-6.000000000000002 + 0.0im,14.999999999999988 + 1.336465207532456
```

In [13]:
```
println(diag(T))
```

```
[-6.000000000000002,14.999999999999988,14.999999999999988]
```

In [14]:
```
Q'*Q, Q*Q'
```

Out[14]:
```
(
3x3 Array{Float64,2}:
 1.0           1.11022e-16  2.77556e-17
 1.11022e-16  1.0           1.52656e-16
 2.77556e-17  1.52656e-16  1.0          ,

3x3 Array{Float64,2}:
 1.0           1.35877e-16  0.0
 1.35877e-16  1.0           3.22346e-17
 0.0           3.22346e-17  1.0          )
```

In [15]:
```
# Similar matrices
M=rand(-5:5,3,3)
B=M*A*inv(M)
eigvals(B), trace(B), det(B)
```

Out[15]:
```
(Complex{Float64}[-6.000000000000011 + 0.0im,14.999999999999954 + 6.930745425917734
```

1.4.4 Example

This matrix is nondefective and diagonalizable.

In [16]:
```
A=[57 -21 21; -14 22 -7; -140 70 -55]
```

Out[16]:
```
3x3 Array{Int64,2}:
   57  -21   21
  -14   22   -7
 -140   70  -55
```

In [17]:
```
p(x)=factor(det(A-x*I))
p(x)
```

Out[17]:

$$-(x-15)^2(x+6)$$

In [18]:
```
λ=solve(p(x),x)
```

Out[18]:

$$\begin{bmatrix}-6\\15\end{bmatrix}$$

In [19]:
```
h=nullspace(A-λ[2]*I)
```

```
Out[19]: 2-element Array{Any,1}:

         [U+23A1]1/2[U+23A4]
         [U+23A2]    [U+23A5]
         [U+23A2] 1 [U+23A5]
         [U+23A2]    [U+23A5]
         [U+23A3] 0 [U+23A6]

         [U+23A1]-1/2[U+23A4]
         [U+23A2]      [U+23A5]
         [U+23A2] 0   [U+23A5]
         [U+23A2]      [U+23A5]
         [U+23A3] 1   [U+23A6]
```

1.4.5 Example

Let us try some random examples of dimension $n = 4$ (the largest n for which we can compute eigevalues symbolically).

```
In [24]: A=rand(-4:4,4,4)
```

```
Out[24]: 4x4 Array{Int64,2}:
           2  -3   0  -1
           4   4   4   4
          -1  -2   2  -4
          -3  -1  -3  -1
```

```
In [25]: p(x)=factor(det(A-x*I))
         p(x)
```

Out[25]:

$$(x-1)\left(x^3-6x^2+15x-16\right)$$

```
In [26]: λ=solve(p(x),x)
```

Out[26]:

$$\left[\begin{array}{c} 1 \\ 2+\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\sqrt{2}}-\frac{1}{\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\sqrt{2}}} \\ 2-\frac{1}{\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\sqrt{2}}}+\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\sqrt{2}} \\ -\frac{1}{\sqrt[3]{1+\sqrt{2}}}+\sqrt[3]{1+\sqrt{2}}+2 \end{array}\right]$$

```
In [27]: length(λ)
```

```
Out[27]: 4
```

Since all eigenvalues are distinct, they are all simple and the matrix is diagonalizable. With high probability, all eigenvalues of a random matrix are simple.
Do not try to use `nullspace()` here.

```
In [28]: A=rand(4,4)
         p(x)=factor(det(A-x*I))
         p(x)
```

Out[28]:

$$\frac{1}{(1.0x - 0.263555114562084)^2 (1.0x^2 - 0.30049008937489x - 0.0273043578708192)} (1.0x^8 - 1.62663978871787x$$

In this case, symbolic computation does not work well with floating-point numbers - the degree of $p_A(x)$ is 8 instead of 4.
Let us try `Rational` numbers:

```
In [29]: A=map(Rational,A)
```

```
Out[29]: 4x4 Array{Rational{Int64},2}:
  296736678933347//1125899906842624  ...    56345917898845//281474976710656
  504995735251837//4503599627370496      1394131076186331//2251799813685248
  286254169702517//2251799813685248      2155038037278727//2251799813685248
  390900094312213//2251799813685248       174231544225239//1125899906842624
```

```
In [30]: p(x)=factor(det(A-x*I))
         p(x)
```

Out[30]:

$$\frac{1}{205688069665150755269371147819668813122841983204197482918576128} (2056880696651507552693711478196$$

```
In [31]: λ=solve(p(x),x)
```

Out[31]:

$$\left[\begin{array}{l} \frac{1799276930165875}{9007199254740992} - \frac{1}{2}\sqrt{\frac{584268513069005690871228300826 09}{6084722881095501127184175385 8048} + \frac{}{740477050794542718969736132150807727242231139535110938506874 06}} \\ \frac{1799276930165875}{9007199254740992} + \frac{1}{2}\sqrt{\frac{58426851306900569087122830082609}{30423614405477505635920876929024} - 2\sqrt[3]{\frac{3621643664469432416240806102707025236780428175626240408 45}{360447812311653576404159912916111919339206388147685154928 5}}} \\ \frac{1799276930165875}{9007199254740992} - \frac{1}{2}\sqrt{\frac{58426851306900569087122830082609}{30423614405477505635920876929024} - 2\sqrt[3]{\frac{362164366446943241624080610270702523678042817562624040845}{3604478123116535764041599129161119193392063881476851549285}}} \\ \frac{1799276930165875}{9007199254740992} + \frac{1}{2}\sqrt{\frac{58426851306900569087122830082609}{30423614405477505635920876929024} - 2\sqrt[3]{\frac{362164366446943241624080610270702523678042817562624040845}{3604478123116535764041599129161119193392063881476851549285}}} \end{array}\right.$$

```
In [32]: length(λ)
```

```
Out[32]: 4
```

1.4.6 Example - Circulant matrix

For more details, see A. Böttcher and I. Spitkovsky, Special Types of Matrices and the references therein.
Given $a_0, a_1, \ldots, a_{n-1} \in \mathbb{C}$, the **circulant matrix** is

$$C(a_0, a_1, \ldots, a_{n-1}) = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_1 \\ a_1 & a_0 & a_{n-1} & \cdots & a_2 \\ a_2 & a_1 & a_0 & \cdots & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{bmatrix}.$$

Let $a(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$ be the associated complex polynomial.
Let $\omega_n = \exp(2\pi i/n)$. The **Fourier matrix** of order n is

$$F_n = \frac{1}{\sqrt{n}} \left[\omega_n^{(j-1)(k-1)}\right]_{j,k=1}^{n} = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots \omega_n^{n-1} & \\ 1 & \omega_n^2 & \omega_n^4 & \cdots \omega_n^{2(n-1)} & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots \omega_n^{(n-1)(n-1)} & \end{bmatrix}.$$

Fourier matrix is unitary. Fourier matrix is Vandermonde matrix, $F_n = \frac{1}{\sqrt{n}} V(1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1})$.
Circulant matrix is normal and, thus, unitarily diagonalizable, with the eigenvalue decompisition

$$C(a_0, a_1, \ldots, a_{n-1}) = F_n^* \operatorname{diag}[(a(1), a(\omega_n), a(\omega_n^2), \ldots, a(\omega_n^{n-1})] F_n.$$

We shall use the package SpecialMatrices.jl.

```
In [33]: using SpecialMatrices
         using Polynomials
```

```
In [34]: whos(SpecialMatrices)
```

```
                 Cauchy    180 bytes  DataType
              Circulant    168 bytes  DataType
              Companion    168 bytes  DataType
              Frobenius    180 bytes  DataType
                 Hankel    168 bytes  DataType
                Hilbert    180 bytes  DataType
                  Kahan    244 bytes  DataType
                Riemann    168 bytes  DataType
         SpecialMatrices  3457 bytes  Module
                 Strang    168 bytes  DataType
               Toeplitz    168 bytes  DataType
            Vandermonde    168 bytes  DataType
```

```
In [35]: n=6
         a=rand(-9:9,n)
```

```
Out[35]: 6-element Array{Int64,1}:
           4
          -5
           8
           4
           5
           2
```

```
In [36]: C=Circulant(a)
```

```
Out[36]: 6x6 SpecialMatrices.Circulant{Int64}:
           4   2   5   4   8  -5
          -5   4   2   5   4   8
           8  -5   4   2   5   4
           4   8  -5   4   2   5
           5   4   8  -5   4   2
           2   5   4   8  -5   4
```

```
In [37]: # Check for normality
         full(C)*full(C)'-full(C)'*full(C)
```

```
Out[37]: 6x6 Array{Int64,2}:
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0
```

```
In [38]: p1=Polynomials.Poly(a)
```

```
Out[38]: Poly(4 - 5x + 8x^2 + 4x^3 + 5x^4 + 2x^5)
```

```
In [39]: ω=exp(2*π*im/n)
```

```
Out[39]: 0.5000000000000001 + 0.8660254037844386im
```

```
In [40]: v=[ω^k for k=0:n-1]
         F=Vandermonde(v)
```

```
Out[40]: 6x6 SpecialMatrices.Vandermonde{Any}:
          1.0+0.0im             1.0+0.0im       ...             1.0+0.0im
          1.0+0.0im             0.5+0.866025im                0.5-0.866025im
          1.0+0.0im            -0.5+0.866025im               -0.5-0.866025im
          1.0+0.0im   -1.0+3.88578e-16im           -1.0+1.94289e-15im
          1.0+0.0im            -0.5-0.866025im               -0.5+0.866025im
          1.0+0.0im             0.5-0.866025im  ...             0.5+0.866025im
```

```
In [41]: Fn=full(F)/sqrt(n)
         Λ=Fn*full(C)*Fn'
```

```
Out[41]: 6x6 Array{Any,2}:
                            18.0+0.0im  ...   1.13243e-14-1.06581e-14im
           -2.44249e-15+2.22045e-16im      -2.66454e-15+1.55431e-15im
           -1.77636e-15+2.27596e-15im       -3.9968e-15-1.11022e-15im
                     0.0+3.4972e-15im       -5.9952e-15-7.10543e-15im
            2.44249e-15+6.21725e-15im      -4.66294e-15-9.10383e-15im
            1.06581e-14+9.65894e-15im   ...                 -8.0+3.4641im
```

```
In [42]: [diag(Λ) p1(v)]
```

```
Out[42]: 6x2 Array{Any,2}:
                   18.0+0.0im                 18.0+0.0im
                   -8.0-3.4641im              -8.0-3.4641im
                    3.0-8.66025im              3.0-8.66025im
          16.0-2.36658e-30im        16.0-7.38298e-15im
                    3.0+8.66025im              3.0+8.66025im
                   -8.0+3.4641im              -8.0+3.4641im
```

1.5 Hermitian and real symmetric matrices

For more details and the proofs of the Facts below, see W. Barrett, Hermitian and Positive Definite Matrices and the references therein.

1.5.1 Definitions

Matrix $A\in\mathbb{C}^{n\times n}$ is **Hermitian** or **self-adjoint** if $A^*=A$, or element-wise, $\bar a_{ij}=a_{ji}$. We say $A\in\mathcal{H}_n$.
Matrix $A\in\mathbb{R}^{n\times n}$ is **symmetric** if $A^T=A$, or element-wise, $a_{ij}=a_{ji}$. We say $A\in\mathcal{S}_n$.
Rayleigh qoutient of $A\in\mathcal{H}_n$ and nonzero vector $x\in\mathbb{C}^n$ is

$$R_A(x)=\frac{x^*Ax}{x^*x}.$$

Matrices $A,B\in\mathcal{H}_n$ are **congruent** if there exists nonsingular matrix C such that $B=C^*AC$.
Inertia of $A\in\mathcal{H}_n$ is the ordered triple

$$\mathrm{in}(A)=(\pi(A),\nu(A),\zeta(A)),$$

where $\pi(A)$ is the number of positive eigenvalues of A, $\nu(A)$ is the number of negative eigenvalues of A, and $\pi(A)$ is the number of zero eigenvalues of A.
Gram matrix of a set of vectors $x_1,x_2,\ldots,x_k\in\mathbb{C}^n$ is the matrix G with entries $G_{ij}=x_i^*x_j$.

1.5.2 Facts

Assume A is Hermitian and $x\in\mathbb{C}^n$ is nonzero. Then

1. Real symmetric matrix is Hermitian, and real Hermitian matrix is symmetric.
2. Hermitian and real symmetric matrices are normal.
3. $A+A^*$, A^*A, and AA^* are Hermitian.
4. If A is nonsingular, A^{-1} is Hermitian.

5. Main diagonal entries of A are real.
6. Matrix T from the Schur decomposition of A is Hermitian. Consequently:
   - T is diagonal and real, and has eigenvalues of A on diagonal,
   - matrix Q of the Schur decomposition is the unitary matrix of eigenvectors,
   - all eigenvalues of A are semisimple and A is nondefective,
   - eigenvectors corresponding to distinct eigenvalues are orthogonal.
7. To summarize *(Spectral Theorem)*:
   - if $A \in \mathcal{H}_n$, there is a unitary matrix U and real diagonal matrix Λ such that $A = U\Lambda U^*$. The diagonal entries of Λ are the eigenvalues of A, and the columns of U are the corresponding eigenvectors.
   - if $A \in \mathcal{S}_n$, the same holds with orthogonal matrix U, $A = U\Lambda U^T$.
   - if $A \in \mathcal{H}_n$ with eigenpairs (λ_i, u_i), then

$$A = \lambda_1 u_1 u_1^* + \lambda_2 u_2 u_2^* + \cdots + \lambda_n u_n u_n^*.$$

   - similarly, if $A \in \mathcal{S}_n$, then

$$A = \lambda_1 u_1 u_1^T + \lambda_2 u_2 u_2^T + \cdots + \lambda_n u_n u_n^T.$$

8. Since all eigenvalues of A are real, they can be ordered:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n.$$

9. *(Rayleigh-Ritz Theorem)* It holds:

$$\begin{aligned} \lambda_n &\leq \frac{x^* A x}{x^* x} \leq \lambda_1, \\ \lambda_1 &= \max_x \frac{x^* A x}{x^* x} = \max_{\|x\|_2 = 1} x^* A x, \\ \lambda_n &= \min_x \frac{x^* A x}{x^* x} = \min_{\|x\|_2 = 1} x^* A x. \end{aligned}$$

10. *(Courant-Fischer Theorem)* Iz holds:

$$\begin{aligned} \lambda_k &= \max_{\dim(W)=k} \min_{x \in W} \frac{x^* A x}{x^* x} \\ &= \min_{\dim(W)=n-k+1} \max_{x \in W} \frac{x^* A x}{x^* x}. \end{aligned}$$

11. *(Cauchy Interlace Theorem)* For $i \in \{1, 2, \ldots, n\}$, let $A(i)$ be the principal submatrix of A obtained by deleting its t-th row and i-th column with ordered eigenvalues $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_{n-1}$. Then

$$\lambda_1 \geq \mu_1 \geq \lambda_2 \geq \mu_2 \geq \lambda_3 \geq \cdots \geq \lambda_{n-1} \geq \mu_{n-1} \geq \lambda_n.$$

12. *(Weyl Inequalities)* For $A, B \in \mathcal{H}_n$, it holds:

$$\begin{aligned} \lambda_{j+k-1}(A+B) &\leq \lambda_j(A) + \lambda_k(B), && \text{for } j+k \leq n+1, \\ \lambda_{j+k-n}(A+B) &\geq \lambda_j(A) + \lambda_k(B), && \text{for } j+k \geq n+1, \end{aligned}$$

and, in particular,

$$\lambda_j(A) + \lambda_n(B) \leq \lambda_j(A+B) \leq \lambda_j(A) + \lambda_1(B), \quad \text{for } j = 1, 2, \ldots, n.$$

13. $\pi(A) + \mu(A) + \zeta(A) = n$.
14. $\operatorname{rank}(A) = \pi(A) + \mu(A)$.
15. If A is nonsingular, $\operatorname{in}(A) = \operatorname{in}(A^{-1})$.
16. If $A, B \in \mathcal{H}_n$ are similar, $\operatorname{in}(A) = \operatorname{in}(B)$.
17. *(Sylvester's Law of Inertia)* $A, B \in \mathcal{H}_n$ are congruent if and only if $\operatorname{in}(A) = \operatorname{in}(B)$.
18. *(Subadditivity of Inertia)* For $A, B \in \mathcal{H}_n$,

$$\pi(A+B) \leq \pi(A) + \pi(B), \qquad \nu(A+B) \leq \nu(A) + \nu(B).$$

19. Gram matrix is Hermitian.

1.5.3 Example - Hermitian matrix

```
In [43]: # Generating Hermitian matrix
         n=5
         A=rand(n,n)+im*rand(n,n)
         A=A+A'
```

```
Out[43]: 5x5 Array{Complex{Float64},2}:
          0.930265+0.0im         0.446557-0.23971im    ...  0.620903+0.558634im
          0.446557+0.23971im     0.792802+0.0im             1.39358-0.0253063im
           1.24079-0.0947762im   0.624038+0.578241im        1.31563-0.10475im
          0.499987-0.306236im     1.07387-0.101868im       0.755314-0.479241im
          0.620903-0.558634im     1.39358+0.0253063im       1.35722+0.0im
```

```
In [44]: ishermitian(A)
```

```
Out[44]: true
```

```
In [45]: # Diagonal entries
         diag(A)
```

```
Out[45]: 5-element Array{Complex{Float64},1}:
          0.930265+0.0im
          0.792802+0.0im
           1.83815+0.0im
          0.865551+0.0im
           1.35722+0.0im
```

In [46]:
```
# Schur decomposition
T,Q=schur(A)
```

Out[46]:
```
(
5x5 Array{Complex{Float64},2}:
 4.97221+2.99221e-16im   ...   3.31972e-16-2.79937e-16im
             0.0+0.0im     -1.70617e-17+3.35373e-16im
             0.0+0.0im      2.72364e-16-9.63298e-17im
             0.0+0.0im      1.24872e-16-1.67455e-16im
             0.0+0.0im         0.169802-1.32495e-17im,

5x5 Array{Complex{Float64},2}:
 -0.351083-1.81092e-16im         ...  -0.374509-1.37831e-11im
          -0.363634+0.153284im              0.129804-0.0787185im
          -0.550146+0.0856471im             0.171122-0.178376im
          -0.332781+0.190726im              -0.17148+0.761714im
          -0.499072+0.121761im              0.252983-0.319347im ,

Complex{Float64}[4.97220930971698 + 2.992206477244406e-16im,-1.1053843410787303 - 2
```

In [47]:
```
λ,U=eig(A)
```

Out[47]:
```
([-1.1053843410787294,0.1698017757791621,0.6180063660633512,1.129358932380346,4.972
5x5 Array{Complex{Float64},2}:
 -0.124628+0.395363im   0.232553+0.293558im   ...  -0.341079-0.0832149im
  0.575162+0.189603im  -0.142305-0.0528654im     -0.389604+0.0627262im
  0.244847-0.278837im  -0.246078-0.0233699im      -0.55477-0.047191im
 -0.269686-0.163431im   0.703548-0.338576im      -0.368504+0.106414im
 -0.473556-0.0im        -0.40741-0.0im           -0.513711-0.0im      )
```

In [48]:
```
# Spectral theorem
A-U*diagm(λ)*U'
```

Out[48]:
```
5x5 Array{Complex{Float64},2}:
  6.66134e-16+2.77556e-17im  ...  0.0+1.11022e-16im
          0.0+7.21645e-16im     0.0+5.55112e-17im
  1.11022e-15+9.71445e-17im     -2.22045e-16+0.0im
 -5.55112e-17+1.11022e-16im      2.22045e-16+0.0im
          0.0-1.11022e-16im       1.9984e-15+0.0im
```

In [49]:
```
# Spectral theorem
A-sum([λ[i]*U[:,i]*U[:,i]' for i=1:n])
```

Out[49]:
```
5x5 Array{Complex{Float64},2}:
 6.66134e-16+1.04083e-17im   1.11022e-16-7.77156e-16im  ...   0.0+2.22045e-16im
         0.0+7.77156e-16im  -5.55112e-16-1.38778e-17im      0.0+1.11022e-16im
 8.88178e-16+1.66533e-16im  -6.66134e-16+2.22045e-16im     -2.22045e-16+0.0im
         0.0+1.11022e-16im   1.11022e-15+3.05311e-16im      2.22045e-16+0.0im
         0.0-1.11022e-16im           0.0-1.11022e-16im       1.9984e-15+0.0im
```

In [50]:
```
λ
```

```
Out[50]: 5-element Array{Float64,1}:
          -1.10538
           0.169802
           0.618006
           1.12936
           4.97221
```

```
In [52]: # Cauchy Interlace Theorem (repeat several times)
         @show i=rand(1:n)
         μ=eigvals(A[[1:i-1;i+1:n],[1:i-1;i+1:n]])
```

```
i = rand(1:n) = 2
```

```
Out[52]: 4-element Array{Float64,1}:
          -0.311907
           0.215373
           0.849404
           4.23832
```

```
In [53]: # Inertia
         inertia(A)=[sum(eigvals(A).>0), sum(eigvals(A).<0), sum(eigvals(A).==0)]
```

```
Out[53]: inertia (generic function with 1 method)
```

```
In [54]: inertia(A)
```

```
Out[54]: 3-element Array{Int64,1}:
          4
          1
          0
```

```
In [55]: # Similar matrices
         C=rand(n,n)+im*rand(n,n)
         inertia(A)
         B=C*A*inv(C)
         inertia(A)==inertia(B)
```

```
        LoadError: MethodError: `isless` has no method matching isless(::Int64, ::Complex{Fl
    Closest candidates are:
      isless(::Real, !Matched::AbstractFloat)
      isless(::Real, !Matched::Real)
      isless(::Integer, !Matched::Char)
      ...
    while loading In[55], in expression starting on line 5


         in bitcache_lt at broadcast.jl:406

         in .< at broadcast.jl:422
```

```
in .> at operators.jl:39

in inertia at In[53]:2
```

This did not work numerically due to rounding errors!

```
In [56]: eigvals(B)

Out[56]: 5-element Array{Complex{Float64},1}:
           4.97221-8.60748e-16im
          -1.10538+2.38106e-16im
           1.12936-2.87045e-15im
          0.618006+8.47049e-16im
          0.169802+2.07205e-15im

In [57]: # Congruent matrices - this does not work either, without some preparation
         B=C'*A*C
         inertia(A)==inertia(B)


        LoadError: MethodError: `isless` has no method matching isless(::Int64, ::Complex{Fl
    Closest candidates are:
      isless(::Real, !Matched::AbstractFloat)
      isless(::Real, !Matched::Real)
      isless(::Integer, !Matched::Char)
      ...
    while loading In[57], in expression starting on line 3



         in bitcache_lt at broadcast.jl:406

         in .< at broadcast.jl:422

         in .> at operators.jl:39

         in inertia at In[53]:2


In [58]: Hermitian(B)


        LoadError: ArgumentError: Cannot construct Hermitian from matrix with nonreal diago
    while loading In[58], in expression starting on line 1



         in call at linalg/symmetric.jl:16

         in call at linalg/symmetric.jl:14
```

In [59]:
```
# We need to symmetrize B so that the right algorithm is called
ishermitian(B), ishermitian((B+B')/2), inertia(A)==inertia((B+B')/2)
```

Out[59]:
```
(false,true,true)
```

In [60]:
```
@which eigvals(B)
```

Out[60]:
```
eigvals{T}(A::Union{DenseArray{T,2},SubArray{T,2,A<:DenseArray{T,N},I<:Tuple{Vararg
```

In [61]:
```
# Weyl Inequalities
B=rand(n,n)+im*rand(n,n)
B=(B+B')/10
@show λ
μ=eigvals(B)
γ=eigvals(A+B)
μ,γ
```

```
λ = [-1.1053843410787294,0.1698017757791621,0.6180063660633512,1.129358932380346,4.97220930
```

Out[61]:
```
([-0.23154741710143145,-0.15326123035607958,0.01796649934153011,0.0828490359329309
```

In [62]:
```
# Theorem uses different order!
j=rand(1:n)
k=rand(1:n)
@show j,k
if j+k<=n+1
    @show sort(γ,rev=true)[j+k-1], sort(λ,rev=true)[j]+sort(μ,rev=true)[k]
end
if j+k>=n+1
    sort(γ,rev=true)[j+k-n], sort(λ,rev=true)[j]+sort(μ,rev=true)[k]
end
```

```
(j,k) = (2,4)
((sort(γ,rev=true))[(j + k) - 1],(sort(λ,rev=true))[j] + (sort(μ,rev=true))[k]) = (-1.24634
```

Out[62]:
```
(5.416728177236637,0.9760977020242663)
```

In [63]:
```
sort(λ,rev=true)
```

Out[63]:
```
5-element Array{Float64,1}:
  4.97221
  1.12936
  0.618006
  0.169802
 -1.10538
```

1.5.4 Example - real symmetric matrix

```
In [64]: # Generating real symmetric matrix
         n=6
         A=rand(-9:9,n,n)
         A=A+A'

Out[64]: 6x6 Array{Int64,2}:
            0   -1  -14    2   -6    5
           -1    2   -3    3  -17   -3
          -14   -3  -14    4  -13    1
            2    3    4  -14    0    0
           -6  -17  -13    0  -16   -8
            5   -3    1    0   -8   -8

In [65]: issym(A)

Out[65]: true

In [66]: T,Q=schur(A)

Out[66]: (
         6x6 Array{Float64,2}:
          -39.7394  -8.80632e-15  -8.10921e-16  ...   1.16954e-15  -1.58972e-15
            0.0     13.6532       -4.44176e-16       -1.47832e-15  -2.9341e-15
            0.0      0.0           9.4661             1.44043e-15   2.41438e-15
            0.0      0.0           0.0                1.42709e-15  -1.0919e-15
            0.0      0.0           0.0              -12.9772       -1.13111e-15
            0.0      0.0           0.0          ...   0.0          -1.82811    ,

         6x6 Array{Float64,2}:
           0.295696   0.102096   0.839836   -0.385488  -0.21092     0.0610468
           0.339894   0.780159  -0.0355622   0.250292   0.127033   -0.442467
           0.555995   0.153619  -0.498047   -0.428417  -0.221213    0.432139
          -0.148995   0.114242  -0.0178641   0.379001  -0.904953    0.0429885
           0.668625  -0.571326   0.0135791   0.367211  -0.0429109  -0.29943
           0.136556   0.133663   0.211792    0.5716     0.263929    0.722666 ,

         [-39.73939736719243,13.653170408610627,9.466101069121056,-18.574569389956697,-12.97

In [67]: λ,U=eig(A)

Out[67]: ([-39.73939736719237,-18.574569389956714,-12.977193295562964,-1.8281114250196318,9
         6x6 Array{Float64,2}:
          -0.295696  -0.385488   0.21092    0.0610468   0.839836   -0.102096
          -0.339894   0.250292  -0.127033  -0.442467   -0.0355622  -0.780159
          -0.555995  -0.428417   0.221213   0.432139   -0.498047   -0.153619
           0.148995   0.379001   0.904953   0.0429885  -0.0178641  -0.114242
          -0.668625   0.367211   0.0429109 -0.29943     0.0135791   0.571326
          -0.136556   0.5716    -0.263929   0.722666    0.211792   -0.133663)

In [68]: A-sum([λ[i]*U[:,i]*U[:,i]' for i=1:n])
```

```
Out[68]: 6x6 Array{Float64,2}:
 -6.63358e-15  -8.21565e-15  -5.32907e-15  …   0.0          -7.99361e-15
 -8.65974e-15   8.88178e-15  -6.66134e-15     7.10543e-15   2.22045e-15
 -5.32907e-15  -5.77316e-15  -1.77636e-14    -3.55271e-15  -2.44249e-15
  9.76996e-15  -8.88178e-16   1.28786e-14     0.0          -4.63518e-15
  8.88178e-16   1.06581e-14  -3.55271e-15     0.0           1.24345e-14
 -9.76996e-15   2.22045e-15  -1.77636e-15  …   1.06581e-14   7.10543e-14
```

```
In [69]: inertia(A)
```

```
Out[69]: 3-element Array{Int64,1}:
 2
 4
 0
```

```
In [70]: C=rand(n,n)
         inertia(C'*A*C)
```

```
Out[70]: 3-element Array{Int64,1}:
 2
 4
 0
```

1.6 Positive definite matrices

These matrices are an important subset of Hermitian or real symmteric matrices.

1.6.1 Definitions

Matrix $A \in \mathcal{H}_n$ is **positive definite** (PD) if $x^*Ax > 0$ for all nonzero $x \in \mathbb{C}^n$.
Matrix $A \in \mathcal{H}_n$ is **positive semidefinite** (PSD) if $x^*Ax \geq 0$ for all nonzero $x \in \mathbb{C}^n$.

1.6.2 Facts

1. $A \in \mathcal{S}_n$ is PD if $x^T Ax > 0$ for all nonzero $x \in \mathbb{R}^n$, and is PSD if $x^T Ax \geq 0$ for all $x \in \mathbb{R}^n$.
2. If $A, B \in \mathrm{PSD}_n$, then $A + B \in \mathrm{PSD}_n$. If, in addition, $A \in \mathrm{PD}_n$, then $A + B \in \mathrm{PD}_n$.
3. If $A \in \mathrm{PD}_n$, then $\mathrm{tr}(A) > 0$ and $\det(A) > 0$.
4. If $A \in \mathrm{PSD}_n$, then $\mathrm{tr}(A) \geq 0$ and $\det(A) \geq 0$.
5. Any principal submatrix of a PD matrix is PD. Any principal submatrix of a PSD matrix is PSD. Consequently, all minors are positive or nonnegative, respectively.
6. $A \in \mathcal{H}_n$ is PD iff *every leading* principal minor of A is positive. $A \in \mathcal{H}_n$ is PSD iff *every* principal minor is nonnegative.
7. For $A \in \mathrm{PSD}_n$, there exists unique PSD k-th **root**, $A^{1/k} = U\Lambda^{1/k}U^*$.
8. *(Cholesky Factorization)* $A \in \mathcal{H}_n$ if PD iff there is an invertible lower triangular matrix L with positive diagonal entries such that $A = LL^*$.
9. Gram matrix is PDS. If the vectors are linearly independent, Gram matrix is PD.

1.6.3 Example - Positive definite matrix

```
In [71]: # Generating positive definite matrix as a Gram matrix
         n=5
         A=rand(n,n)+im*rand(n,n)
         A=A*A'

Out[71]: 5x5 Array{Complex{Float64},2}:
          2.13672+0.0im       2.04993+0.652056im    ...  1.76093+0.404015im
          2.04993-0.652056im  3.04994+0.0im            1.66653-0.00587273im
          2.41067-0.723804im  2.82863-0.0383012im       2.7673+0.0468083im
          2.61585-0.995044im  3.14551+0.168999im       2.53888+0.0152369im
          1.76093-0.404015im  1.66653+0.00587273im     2.31123+0.0im

In [72]: ishermitian(A)

Out[72]: true

In [73]: eigvals(A)

Out[73]: 5-element Array{Float64,1}:
           0.0415939
           0.181826
           0.624074
           1.15728
          13.9352

In [74]: # Positivity of principal leading minors
         [det(A[1:i,1:i]) for i=1:n]

Out[74]: 5-element Array{Any,1}:
                     2.13672+0.0im
            1.88947-3.33067e-16im
            1.52243-5.55112e-17im
            0.73343-1.66533e-16im
          0.0761158-9.19403e-17im

In [75]: # Square root
         λ,U=eig(A)
         Ar=U*diagm(sqrt(λ))*U'
         A-Ar*Ar

Out[75]: 5x5 Array{Complex{Float64},2}:
                    0.0-5.55112e-17im  ...  2.22045e-16-1.66533e-16im
           2.22045e-15-7.77156e-16im     6.66134e-16+2.77556e-17im
           4.44089e-16-7.77156e-16im             0.0-4.16334e-17im
          -8.88178e-16-5.55112e-16im                     0.0+0.0im
          -2.22045e-16+2.22045e-16im             0.0+6.93889e-18im

In [76]: # Cholesky factorization - the upper triangular factor is returned
         L=chol(A)
```

```
Out[76]: 5x5 UpperTriangular{Complex{Float64},Array{Complex{Float64},2}}:
          1.46175+0.0im   1.40238+0.446078im  ...    1.20467+0.276391im
              0.0+0.0im  0.940364+0.0im            -0.155439+0.153025im
              0.0+0.0im       0.0+0.0im             0.757055+0.140758im
              0.0+0.0im       0.0+0.0im             0.103922+0.168838im
              0.0+0.0im       0.0+0.0im              0.32215+0.0im
```

```
In [77]: A-L'*L
```

```
Out[77]: 5x5 Array{Complex{Float64},2}:
          4.44089e-16+0.0im  0.0+0.0im  ...  0.0+0.0im            0.0+0.0im
                  0.0+0.0im  0.0+0.0im       0.0+0.0im            0.0+0.0im
                  0.0+0.0im  0.0+0.0im       0.0+0.0im  -4.44089e-16+0.0im
                  0.0+0.0im  0.0+0.0im       0.0+0.0im            0.0+0.0im
                  0.0+0.0im  0.0+0.0im       0.0+0.0im            0.0+0.0im
```

1.6.4 Example - Positive semidefinite matrix

```
In [78]: # Generating positive semidefinite matrix as a Gram matrix, try it several times
         n=6
         m=4
         A=rand(-9:9,n,m)
```

```
Out[78]: 6x4 Array{Int64,2}:
          -8   1  -2  -8
           0   2  -9   3
           4  -4  -3  -4
           8   8  -5   1
          -6   4  -3   1
          -3   7  -5  -5
```

```
In [79]: A=A*A'
```

```
Out[79]: 6x6 Array{Int64,2}:
          133  -4    2  -54   50   81
           -4  94    7   64   38   44
            2   7   57   11  -35   -5
          -54  64   11  154    0   52
           50  38  -35    0   62   56
           81  44   -5   52   56  108
```

```
In [80]: # There are rounding errors!
         eigvals(A)
```

```
Out[80]: 6-element Array{Float64,1}:
           -4.55977e-14
            3.47453e-14
           53.5483
           79.3895
          222.5
          252.562
```

```
In [81]: # Cholesky factorization - this can fail
         L=chol(A)
```

```
LoadError: Base.LinAlg.PosDefException(6)
while loading In[81], in expression starting on line 2


    in chol! at linalg/cholesky.jl:28

    in chol at linalg/cholesky.jl:83
```

1.6.5 Example - Covariance and corellation matrices

Covariance and correlation matrices are PSD.
Correlation matrix is diagonally scaled covariance matrix.

```
In [82]: x=rand(10,5)
```

```
Out[82]: 10x5 Array{Float64,2}:
          0.382984  0.811281     0.278149   0.974734  0.729655
          0.310468  0.0102155    0.56755    0.176882  0.0530882
          0.984338  0.236825     0.865804   0.487037  0.50051
          0.253985  0.589349     0.770382   0.197951  0.235257
          0.956739  0.857433     0.285229   0.319533  0.88351
          0.252092  0.796644     0.172707   0.796444  0.539951
          0.608766  0.911805     0.311762   0.163555  0.940078
          0.536686  0.000851031  0.161175   0.997979  0.732685
          0.26581   0.905121     0.0715346  0.556449  0.881676
          0.415975  0.568633     0.682676   0.824884  0.727126
```

```
In [83]: A=cov(x)
```

```
Out[83]: 5x5 Array{Float64,2}:
           0.0766409   -0.00836436   0.0173629  -0.0154941    0.0267039
          -0.00836436   0.13        -0.0400246  -0.00779476   0.0629573
           0.0173629   -0.0400246    0.0790824  -0.0334204   -0.0477527
          -0.0154941   -0.00779476  -0.0334204   0.109666     0.0309496
           0.0267039    0.0629573   -0.0477527   0.0309496    0.0851784
```

```
In [84]: B=cor(x)
```

```
Out[84]: 5x5 Array{Float64,2}:
           1.0        -0.0837976   0.223024  -0.169006    0.330507
          -0.0837976   1.0        -0.394744  -0.0652824   0.598288
           0.223024   -0.394744    1.0       -0.358869   -0.581826
          -0.169006   -0.0652824  -0.358869   1.0         0.320224
           0.330507    0.598288   -0.581826   0.320224    1.0
```

```
In [85]: # Diagonal scaling
         D=1./sqrt(diag(A))
```

```
Out[85]: 5-element Array{Float64,1}:
          3.61218
          2.77351
          3.55599
          3.0197
          3.42638

In [86]: diagm(D)*A*diagm(D)

Out[86]: 5x5 Array{Float64,2}:
           1.0        -0.0837976   0.223024  -0.169006    0.330507
          -0.0837976   1.0        -0.394744  -0.0652824   0.598288
           0.223024   -0.394744    1.0       -0.358869   -0.581826
          -0.169006   -0.0652824  -0.358869   1.0         0.320224
           0.330507    0.598288   -0.581826   0.320224    1.0

In [87]: eigvals(A)

Out[87]: 5-element Array{Float64,1}:
          0.00983091
          0.0408597
          0.091886
          0.126605
          0.211385

In [88]: eigvals(B)

Out[88]: 5-element Array{Float64,1}:
          0.110847
          0.435094
          0.997431
          1.28431
          2.17232

In [89]: C=cov(x')

Out[89]: 10x10 Array{Float64,2}:
           0.0865769    -0.0535646   -0.0758885   ...   0.0846432   0.0249248
          -0.0535646     0.0506594    0.0561362        -0.0804714  -0.00326197
          -0.0758885     0.0561362    0.092953         -0.0995166  -0.01967
          -0.036562      0.0279582    0.00301951       -0.0309425  -0.00422148
          -0.000188678  -0.0418649   -0.0102708         0.0592671  -0.0350109
           0.0840092    -0.0545177   -0.0820834   ...   0.0870902   0.0196161
           0.00988806   -0.0506268   -0.0425159         0.0888104  -0.021833
           0.0537261    -0.0174446    0.00674174        0.0134598   0.0313896
           0.0846432    -0.0804714   -0.0995166         0.136223    0.0120651
           0.0249248    -0.00326197  -0.01967           0.0120651   0.0247001

In [90]: eigvals(C)

Out[90]: 10-element Array{Float64,1}:
          -3.90575e-17
```

```
-1.67429e-17
-9.14431e-18
 9.04107e-18
 1.40744e-17
 6.38284e-17
 0.0241912
 0.163102
 0.283797
 0.468208
```

```
In [ ]:
```

2 Eigenvalue Decomposition - Perturbation Theory

2.1 Prerequisites

The reader should be familiar with basic linear algebra concepts and facts about eigenvalue decomposition.

2.2 Competences

The reader should be able to understand and check the facts about perturbations of eigenvalues and eigenvectors.

2.3 Norms

In order to measure changes, we need to define norms. For more details and the proofs of the Facts below, see R. Byers and B. N. Datta, Vector and Matrix Norms, Error Analysis, Efficiency, and Stability and the references therein.

2.3.1 Definitions

Norm on a vector space X is a real-valued function $\| \ \| : X \to \mathbb{R}$ with the following properties:

1. $\|x\| \geq 0$ and $\|x\| = 0$ if and only if x is the zero vector *(Positive definiteness)*
2. $\|\lambda x\| = |\lambda| \|x\|$ *(Homogeneity)*
3. $\|x + y\| \leq \|x\| + \|y\|$ *(Triangle inequality)*

Commonly encountered vector norms for $x \in \mathbb{C}^n$ are:

- **Hölder norm** or p-**norm**: for $p \geq 1$, $\|x\|_p = \left(|x_1|^p + |x_2|^p + \cdots |x_n|^p\right)^{1/p}$,
- **Sum norm** or 1-**norm**: $\|x\|_1 = |x_1| + |x_2| + \cdots |x_n|$,
- **Euclidean norm** or 2-**norm**: $\|x\|_2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots |x_n|^2}$,
- **Sup-norm** or ∞-**norm**: $\|x\|_\infty = \max\limits_{i=1,\ldots,n} |x_i|$.

Vector norm is **absolute** if $\||x|\| = \|x\|$.
Vector norm is **monotone** if $|x| \leq |y|$ implies $\|x\| \leq \|y\|$.
From every vector norm we can derive a corresponding **induced** matrix norm (also, **operator norm** or **natural norm**):

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|.$$

For matrix $A \in \mathbb{C}^{m \times n}$ we define:

- **Maximum absolute column sum norm**: $\|A\|_1 = \max\limits_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$,
- __ Spectral norm__: $\|A\|_2 = \sqrt{\rho(A^* A)} = \sigma_{\max}(A)$ (largest singular value of A),
  - **Maximum absolute row sum norm**: $\|A\|_\infty = \max\limits_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$,

- **Euclidean norm** or **Frobenius norm**: $\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2} = \sqrt{\mathrm{tr}(A^* A)}$.

Matrix norm is **consistent** if $\|A \cdot B\| \leq \|A\| \cdot \|B\|$, where A and B are compatible for matrix multiplication.
Matrix norm is **absolute** if $\||A|\| = \|A\|$.

2.3.2 Examples

```
In [1]: x=rand(-4:4,5)

Out[1]: 5-element Array{Int64,1}:
         -1
          0
          0
          3
          1

In [2]: norm(x,1), norm(x), norm(x,Inf)

Out[2]: (5.0,3.3166247903554,3.0)

In [3]: A=rand(-4:4,7,5)

Out[3]: 7x5 Array{Int64,2}:
          3   2   1   4   3
          0   0   4   1  -2
         -3  -1   0  -1   4
          3  -2   0   1  -3
         -1   4  -1   1  -3
          1  -4   1   4   3
          4   4   0  -1   3

In [4]: norm(A,1), norm(A), norm(A,2), norm(A,Inf), vecnorm(A)

Out[4]: (21.0,8.630367474712905,8.630367474712905,13.0,14.933184523068078)
```

2.3.3 Facts

1. $\|x\|_1$, $\|x\|_2$, $\|x\|_\infty$ and $\|x\|_p$ are absolute and monotone vector norms.
2. A vector norm is absolute iff it is monotone.
3. *Convergence*: $x_k \to x_*$ iff for any vector norm $\|x_k - x_*\| \to 0$.
4. Any two vector norms are equivalent in the sense that, for all x and some $\alpha, \beta > 0$

$$\alpha \|x\|_\mu \leq \|x\|_\nu \leq \beta \|x\|_\mu.$$

   In particular:

- $\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2$,
- $\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n}\|x\|_\infty$,
- $\|x\|_\infty \leq \|x\|_1 \leq n\|x\|_\infty$.

2. *Cauchy-Schwartz inequality*: $|x^* y| \leq \|x\|_2 \|y\|_2$.

3. *Hölder inequality*: if $p, q \geq 1$ and $\frac{1}{p} + \frac{1}{q} = 1$, then $|x^* y| \leq \|x\|_p \|y\|_q$.
4. $\|A\|_1$, $\|A\|_2$ and $\|A\|_\infty$ are induced by the corresponding vector norms.
5. $\|A\|_F$ is not and induced norm.
6. $\|A\|_1$, $\|A\|_2$, $\|A\|_\infty$ and $\|A\|_F$ are consistent.
7. $\|A\|_1$, $\|A\|_\infty$ and $\|A\|_F$ are absolute. However, $\||A|\|_2 \neq \|A\|_2$.
8. Any two matrix norms are equivalent in the sense that, for all A and some $\alpha, \beta > 0$

$$\alpha \|A\|_\mu \leq \|A\|_\nu \leq \beta \|A\|_\mu.$$

In particular:

- $\frac{1}{\sqrt{n}} \|A\|_\infty \leq \|A\|_2 \leq \sqrt{m} \|A\|_\infty$,
- $\|A\|_2 \leq \|A\|_F \leq \sqrt{n} \|A\|_2$,
- $\frac{1}{\sqrt{m}} \|A\|_1 \leq \|A\|_2 \leq \sqrt{n} \|A\|_1$.

6. $\|A\|_2 \leq \sqrt{\|A\|_1 \|A\|_\infty}$.
7. $\|AB\|_F \leq \|A\|_F \|B\|_2$ and $\|AB\|_F \leq \|A\|_2 \|B\|_F$.
8. If $A = xy^*$, then $\|A\|_2 = \|A\|_F = \|x\|_2 \|y\|_2$.
9. $\|A^*\|_2 = \|A\|_2$ and $\|A^*\|_F = \|A\|_F$.
10. For a unitary matrix U of compatible dimension,

$$\|AU\|_2 = \|A\|_2, \quad \|AU\|_F = \|A\|_F, \quad \|UA\|_2 = \|A\|_2, \quad \|UA\|_F = \|A\|_F.$$

11. For A square, $\rho(A) \leq \|A\|$.
12. For A square, $A_k \to 0$ iff $> \rho(A) < 1$.

In [5]:
```
# Absolute norms
norm(A,1), norm(abs(A),1), norm(A,Inf), norm(abs(A),Inf), vecnorm(A), vecnorm(abs(A
```

Out[5]:
```
(21.0,21.0,13.0,13.0,14.933184523068078,14.933184523068078,8.630367474712905,13.343
```

In [6]:
```
# Equivalence of norms
m,n=size(A)
norm(A,Inf)\sqrt(n),norm(A), sqrt(m)*norm(A,Inf)
```

Out[6]:
```
(0.17200522903844537,8.630367474712905,34.39476704383968)
```

In [7]:
```
norm(A), vecnorm(A), sqrt(n)*norm(A)
```

Out[7]:
```
(8.630367474712905,14.933184523068078,19.298088344261252)
```

In [8]:
```
norm(A,1)\sqrt(m),norm(A), sqrt(n)*norm(A,1)
```

Out[8]:
```
(0.12598815766974242,8.630367474712905,46.95742752749558)
```

In [9]:
```
# Fact 12
norm(A), sqrt(norm(A,1)*norm(A,Inf))
```

Out[9]:
```
(8.630367474712905,16.522711641858304)
```

In [10]:
```
# Fact 13
@show B=rand(n,rand(1:9))
vecnorm(A*B), vecnorm(A)*norm(B), norm(A)*vecnorm(B)
```

```
B = rand(n,rand(1:9)) = [0.14495313989355973 0.862559195282494 0.01605065932080363 0.8401625
 0.9453842917188175 0.6920182886342445 0.15450380085344428 0.5418628642600443 0.721535160867
 0.9908564135480644 0.16589717593352749 0.6633695737378855 0.9327887269547015 0.626916510516
 0.9131840076398012 0.19744750171034164 0.08025578819472812 0.06845316582414762 0.214215983
 0.21266906462893442 0.12458959053072438 0.6112940786953798 0.7620920738383359 0.70189323605
```

```
Out[10]: (20.551011657783857,39.51029023753674,25.680600310151107)

In [11]: # Fact 14
         x=rand(10)+im*rand(10)
         y=rand(10)+im*rand(10)
         A=x*y'
         norm(A), vecnorm(A), norm(x)*norm(y)

Out[11]: (6.534720162699072,6.534720162699072,6.534720162699073)

In [12]: # Fact 15
         A=rand(-4:4,7,5)+im*rand(-4:4,7,5)
         norm(A), norm(A'), vecnorm(A), vecnorm(A')

Out[12]: (14.818932189561746,14.818932189561748,19.974984355438178,19.974984355438178)

In [13]: # Unitary invariance - generate random unitary matrix U
         U,R=qr(rand(size(A))+im*rand(size(A)),thin=false)

Out[13]: (
         7x7 Array{Complex{Float64},2}:
           -0.242441-0.348882im       0.157628-0.195961im   ...  -0.233571-0.310607im
          -0.0957171-0.0514461im   -0.00987517-0.337264im        0.192149+0.148996im
          -0.0539785-0.216606im      -0.311047-0.0264378im      -0.304321+0.322752im
             -0.2287-0.389349im     3.83584e-5+0.238172im       -0.123098-0.108093im
           -0.265665-0.278883im      -0.397433-0.233611im       -0.140684+0.289424im
           -0.300869-0.396274im      -0.229445+0.243669im   ...   0.283998-0.402601im
           -0.397524-0.00569077im     0.312576-0.49647im            0.4689+0.00781712im,

         5x5 Array{Complex{Float64},2}:
          -2.48169+0.0im   -1.99446-0.260571im  ...   -1.35313-0.0937606im
               0.0+0.0im  -0.972279+0.0im            -0.915097+0.09813im
               0.0+0.0im        0.0+0.0im            -0.367113-0.598992im
               0.0+0.0im        0.0+0.0im            -0.647959-0.430968im
               0.0+0.0im        0.0+0.0im             0.762635+0.0im      )

In [14]: norm(A), norm(U*A), vecnorm(A), vecnorm(U*A)

Out[14]: (14.818932189561746,14.81893218956175,19.974984355438178,19.97498435543818)

In [15]: # Spectral radius
         A=rand(7,7)+im*rand(7,7)
         maxabs(eigvals(A)), norm(A,Inf), norm(A,1), norm(A), vecnorm(A)

Out[15]: (5.03153499805523,6.185218050341691,6.557337870559409,5.183551400879956,5.69050866
```

```
In [16]: # Fact 18
         B=A/(maxabs(eigvals(A))+2)
         @show maxabs(eigvals(B))
         B^20
```

```
maxabs(eigvals(B)) = 0.7155670844910607
```

```
Out[16]: 7x7 Array{Complex{Float64},2}:
                   -1.84667e-5+0.000101599im  ...  -5.99373e-5+0.000152871im
                   -1.82526e-5+0.000119428im       -6.5065e-5+0.000180757im
                    -1.0326e-5+0.000135897im      -5.77417e-5+0.000208882im
                   -2.05565e-5+0.000110855im      -6.60335e-5+0.000166673im
          -3.21678e-5+7.97822e-5im                -7.46326e-5+0.000114633im
                    1.25282e-5+0.000136569im  ...  -2.22871e-5+0.000216931im
                    2.88948e-5+0.000120276im        8.2413e-6+0.000196522im
```

2.4 Errors and condition numbers

We want to answer the question:
How much the value of a function changes with respect to the change of its argument?

2.4.1 Definitions

For function $f(x)$ and argument x, the **absolute error** with respect to the **perturbation** of the argument δx is

$$\|f(x+\delta x)-f(x)\| \leq \frac{\|f(x+\delta x)-f(x)\|}{\|\delta x\|}\|\delta x\| \equiv \kappa\|\delta x\|.$$

The **condition** or **condition number** κ tells how much does the perturbation of the argument increase. (Its form resembles derivative.)
Similarly, the **relative error** with respect to the relative perturbation of the argument is

$$\frac{\|f(x+\delta x)-f(x)\|}{\|f(x)\|} \leq \frac{\|f(x+\delta x)-f(x)\|\cdot\|x\|}{\|\delta x\|\cdot\|f(x)\|}\cdot\frac{\|\delta x\|}{\|x\|} \equiv \kappa_{rel}\frac{\|\delta x\|}{\|x\|}.$$

2.5 Peturbation bounds

2.5.1 Definitions

Let $A \in \mathbb{C}^{n\times n}$.
Pair $(\lambda, x) \in \mathbb{C}\times\mathbb{C}^{n\times n}$ is an **eigenpair** of A if $x \neq 0$ and $Ax = \lambda x$.
Triplet $(y, \lambda, x) \in \times\mathbb{C}^n\times\mathbb{C}\times\mathbb{C}^n$ is an **eigentriplet** of A if $x, y \neq 0$ and $Ax = \lambda x$ and $y^*A = \lambda y^*$.
Eigenvalue matrix is a diagonal matrix $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$.
If all eigenvalues are real, they can be increasingly ordered. $\Lambda^{\uparrow}$ is the eigenvalue matrix of increasingly ordered eigenvalues.
τ is a **permutation** of $\{1, 2, \ldots, n\}$.
$\tilde{A} = A + \Delta A$ is a **perturbed matrix**, where ΔA is **perturbation**. $(\tilde{\lambda}, \tilde{x})$ are the eigenpairs of $\tilde{A}$.

Condition number of a nonsingular matrix X is $\kappa(X) = \|X\|\|X^{-1}\|$.
Let $X, Y \in \mathbb{C}^{n\times k}$ with $\text{rank}(X) = \text{rank}(Y) = k$. The **canonical angles** between their column spaces are $\theta_i = \cos^{-1}\sigma_i$, where σ_i are the singular values of $(Y^*Y)^{-1/2}Y^*X(X^*X)^{-1/2}$. The **canonical angle matrix** between X and Y is

$$\Theta(X,Y) = \text{diag}(\theta_1, \theta_2, \ldots, \theta_k).$$

2.5.2 Facts

Bounds become more strict as matrices have more structure. Many bounds have versions in spectral norm and Frobenius norm. For more details and the proofs of the Facts below, see R.-C. Li, Matrix Perturbation Theory, and the references therein.

1. There exists τ such that
$$\|\Lambda - \tilde{\Lambda}_\tau\|_2 \le 4(\|A\|_2 + \|\tilde{A}\|_2)^{1-1/n}\|\Delta A\|_2^{1/n}.$$
2. *(First-order perturbation bounds)* Let (y, λ, x) be an eigentriplet of a simple λ. ΔA changes λ to $\tilde{\lambda} = \lambda + \delta\lambda$, where
$$\delta\lambda = \frac{y^*(\Delta A)x}{y^*x} + O(\|\Delta A\|_2^2).$$
3. Let λ be a semisimple eigenvalue of A with multiplicitiy k, and let $X, Y \in \mathbb{C}^{n\times k}$ be the matrices of the corresponding right and left eigenvectors, that is, $AX = \lambda X$ and $Y^*A = \lambda Y^*$, such that $Y^*X = I_k$. ΔA changes the k copies of μ to $\tilde{\mu} = \mu + \delta\mu_i$, where $\delta\mu_i$ are the eigenvalues of $Y^*(\Delta A)X$ up to $O(\|\Delta A\|_2^2$.
4. *(Bauer-Fike Theorem)* If A is diagonalizable and $A = X\Lambda X^{-1}$ is its eigenvalue decomposition, then
$$\max_i \min_j |\tilde{\lambda}_i - \lambda_j| \le \|X^{-1}(\Delta A)X\|_p \le \kappa_p(X)\|\Delta A\|_p.$$
5. If A and $\tilde{A}$ are diagonalizable, there exists τ such that
$$\|\Lambda - \tilde{\Lambda}_\tau\|_F \le \sqrt{\kappa_2(X)\kappa_2(\tilde{X})}\|\Delta A\|_F.$$
If Λ and $\tilde{\Lambda}$ are real, then
$$\|\Lambda^\uparrow - \tilde{\Lambda}^\uparrow\|_{2,F} \le \sqrt{\kappa_2(X)\kappa_2(\tilde{X})}\|\Delta A\|_{2,F}.$$
6. If A is normal, there exists τ such that $\|\Lambda - \tilde{\Lambda}_\tau\|_F \le \sqrt{n}\|\Delta A\|_F$.
7. *(Hoffman-Wielandt Theorem)* If A and $\tilde{A}$ are normal, there exists τ such that $\|\Lambda - \tilde{\Lambda}_\tau\|_F \le \|\Delta A\|_F$.
8. If A is Hermitian, for any unitarily invariant norm $\|\Lambda^\uparrow - \tilde{\Lambda}^\uparrow\| \le \|\Delta A\|$. In particular,
$$\max_i |\lambda_i^\uparrow - \tilde{\lambda}_i^\uparrow| \le \|\Delta A\|_2,$$
$$\sqrt{\sum_i (\lambda_i^\uparrow - \tilde{\lambda}_i^\uparrow)^2} \le \|\Delta A\|_F.$$

9. *(Residual bounds)* Let A be Hermitian. For some $\tilde\lambda \in \mathbb{R}$ and $\tilde x \in \mathbb{C}^n$ with $\|\tilde x\|_2 = 1$, define **residual** $r = A\tilde x - \tilde\lambda \tilde x$. Then $|\tilde\lambda - \lambda| \leq \|r\|_2$ for some $\lambda \in \sigma(A)$.

10. Let, in addition, $\tilde\lambda = \tilde x^* A \tilde x$, let λ be closest to $\tilde\lambda$ and x be its unit eigenvector, and let
$$\eta = \operatorname{gap}(\tilde\lambda) = \min_{\lambda\neq\mu\in\sigma(A)} |\tilde\lambda - \mu|.$$
If $\eta > 0$, then
$$|\tilde\lambda - \lambda| \leq \frac{\|r\|_2^2}{\eta}, \quad \sin\theta(x,\tilde x) \leq \frac{\|r\|_2}{\eta}.$$

11. Let A be Hermitian, $X \in \mathbb{C}^{n\times k}$ have full column rank, and $M \in \mathcal{H}_k$ having eigenvalues $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_k$. Set $R = AX - XM$. Then there exist $\lambda_{i_1} \leq \lambda_{i_2} \leq \cdots \leq \lambda_{i_k} \in \sigma(A)$ such that
$$\max_{1\leq j\leq k} |\mu_j - \lambda_{i_j}| \leq \frac{\|R\|_2}{\sigma_{\min}(X)},$$
$$\sqrt{\sum_{j=1}^k (\mu_j - \lambda_{i_j})^2} \leq \frac{\|R\|_F}{\sigma_{\min}(X)}.$$
(The indices i_j need not be the same in the above formulae.)

12. If, additionally, $X^*X = I$ and $M = X^*AX$, and if all but k of A's eigenvalues differ from every one of M's eigenvalues by at least $\eta > 0$, then
$$\sqrt{\sum_{j=1}^k (\mu_j - \lambda_{i_j})^2} \leq \frac{\|R\|_F^2}{\eta\sqrt{1 - \|R\|_F^2/\eta^2}}.$$

13. Let $A = \begin{bmatrix} M & E^* \\ E & H \end{bmatrix}$ and $\tilde A = \begin{bmatrix} M & 0 \\ 0 & H \end{bmatrix}$ be Hermitian, and set $\eta = \min|\mu - \nu|$ over all $\mu \in \sigma(M)$ and $\nu \in \sigma(H)$. Then
$$\max|\lambda_j^\uparrow - \tilde\lambda_j^\uparrow| \leq \frac{2\|E\|_2^2}{\eta + \sqrt{\eta^2 + 4\|E\|_2^2}}.$$

14. Let
$$\begin{bmatrix} X_1^* \\ X_2^* \end{bmatrix} A \begin{bmatrix} X_1 & X_2 \end{bmatrix} = \begin{bmatrix} A_1 & \\ & A_2 \end{bmatrix}, \quad \begin{bmatrix} X_1 & X_2 \end{bmatrix} \text{unitary}, \quad X_1 \in \mathbb{C}^{n\times k}.$$
Let $Q \in \mathbb{C}^{n\times k}$ have orthonormal columns and for a Hermitian $k\times k$ matrix M set $R = AQ - QM$. Let $\eta = \min|\mu - \nu|$ over all $\mu \in \sigma(M)$ and $\nu \in \sigma(A_2)$. If $\eta > 0$, then
$$\|\sin\Theta(X_1, Q)\|_F \leq \frac{\|R\|_F}{\eta}.$$

2.5.3 Example - Nondiagonalizable matrix

```
In [17]: A=[-3 7 -1; 6 8 -2; 72 -28 19]
```

```
Out[17]: 3x3 Array{Int64,2}:
          -3    7  -1
           6    8  -2
          72  -28  19
```

```
In [18]: # (Right) eigenvectors
         λ,X=eig(A)

Out[18]: ([-6.000000000000005,15.000000241477958,14.999999758522048],
         3x3 Array{Float64,2}:
           0.235702   0.218218  -0.218218
          -0.235702   0.436436  -0.436436
          -0.942809  -0.872872   0.872872)

In [19]: cond(X)

Out[19]: 9.091581949434164e7

In [20]: # Left eigenvectors
         λ1,Y=eig(A')

Out[20]: (Complex{Float64}[-5.999999999999998 + 0.0im,14.999999999999993 + 2.00882626072141
         3x3 Array{Complex{Float64},2}:
              0.894427+0.0im                 0.970143+0.0im                 0.970143-0.0im
             -0.447214+0.0im   -7.58506e-16-1.62404e-8im    -7.58506e-16+1.62404e-8im
          -6.07504e-17+0.0im       0.242536+4.0601e-9im         0.242536-4.0601e-9im   )

In [21]: # Try k=2,3
         k=1
         Y[:,k]'*A-λ[k]*Y[:,k]'

Out[21]: 1x3 Array{Complex{Float64},2}:
          8.88178e-16+0.0im  8.88178e-16+0.0im  -1.7408e-15+0.0im

In [22]: ΔA=rand(3,3)/20
         B=A+ΔA

Out[22]: 3x3 Array{Float64,2}:
          -2.95663    7.03661  -0.995273
           6.00513    8.04044  -1.96629
          72.0063   -27.9913   19.0024

In [23]: μ,Z=eig(B)

Out[23]: (Complex{Float64}[-5.951157189019204 + 0.0im,15.018700806403988 + 0.63185030880100
         3x3 Array{Complex{Float64},2}:
          -0.235969+0.0im  -0.213566+0.0340179im  -0.213566-0.0340179im
           0.233829+0.0im  -0.424639+0.0677232im  -0.424639-0.0677232im
           0.943209+0.0im   0.876543+0.0im         0.876543-0.0im       )

In [24]: # Fact 2
         δλ=μ[k]-λ[k]

Out[24]: 0.04884281098080123 + 0.0im

In [25]: Y[:,k]'*ΔA*X[:,k]/(Y[:,k]⋅X[:,k])

Out[25]: 1-element Array{Complex{Float64},1}:
          0.048619+0.0im
```

2.5.4 Example - Jordan form

In [26]:
```
n=70
c=0.5
J=Bidiagonal(c*ones(n),ones(n-1),true)
```

Out[26]:
```
70x70 Bidiagonal{Float64}:
 0.5  1.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.5  1.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.5  1.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.5  1.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.5  1.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.5  1.0  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.5  1.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮                ⋱                ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.5  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.5  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.5  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.5  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.5  1.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.5  1.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.5
```

In [27]:
```
# Accurately defined eigenvalues
λ=eigvals(J)
```

Out[27]:
```
70-element Array{Float64,1}:
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 ⋮
```

```
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
 0.5
```

```
In [28]: # Only one eigenvector
         X=eigvecs(J)
```

```
Out[28]: 70x70 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱            ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
In [29]: x=eigvecs(J)[:,1]
         y=eigvecs(J')[:,1]
```

```
Out[29]: 70-element Array{Float64,1}:
 0.0
 0.0
 0.0
```

```
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
⋮
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
1.0
```

```
In [30]: y'*full(J)-0.5*y'
```

```
Out[30]: 1x70 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
In [31]: # Just one perturbed element in the lower left corner
         ΔJ=sqrt(eps())*[zeros(n-1);1]*eye(1,n)
```

```
Out[31]: 70x70 Array{Float64,2}:
 0.0          0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                                ⋮    ⋱                 ⋮
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.49012e-8   0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
In [32]: B=J+ΔJ
         μ=eigvals(B)
```

```
Out[32]: 70-element Array{Complex{Float64},1}:
          -0.273017+0.0im
          -0.269905+0.0692927im
          -0.269905-0.0692927im
          -0.260594+0.138027im
          -0.260594-0.138027im
          -0.245159+0.205651im
          -0.245159-0.205651im
          -0.223725+0.271619im
          -0.223725-0.271619im
          -0.196464+0.335399im
          -0.196464-0.335399im
          -0.163595+0.39648im
          -0.163595-0.39648im
                   ⋮
            1.26059+0.138027im
            1.26059-0.138027im
            1.24516+0.205651im
            1.24516-0.205651im
            1.22373+0.271619im
            1.22373-0.271619im
            1.19646+0.335399im
            1.19646-0.335399im
             1.1636+0.39648im
             1.1636-0.39648im
            1.12538+0.454368im
            1.12538-0.454368im
```

```
In [33]: # Fact 2
         maximum(abs(λ-μ))
```

```
Out[33]: 0.7730166686363998
```

```
In [34]: y'*ΔJ*x/(y⋅x)
```

```
Out[34]: 1-element Array{Float64,1}:
          Inf
```

However, since B is diagonalizable, we can apply Bauer-Fike theorem to it:

```
In [35]: μ,Y=eig(B)
```

```
Out[35]: (Complex{Float64}[-0.273017+0.0im,-0.269905+0.0692927im,-0.269905-0.0692927im,-0.26
         70x70 Array{Complex{Float64},2}:
            -0.634386+0.0im  ...              0.634386-0.0im
             0.490391+0.0im                 0.396734-0.288244im
             -0.37908+0.0im                 0.117142-0.360527im
             0.293035+0.0im               -0.0905529-0.278693im
            -0.226521+0.0im                -0.183259-0.133146im
             0.175105+0.0im  ...    -0.175105-8.85556e-13im
            -0.135359+0.0im                -0.109508+0.0795619im
             0.104635+0.0im               -0.0323339+0.0995134im
           -0.0808843+0.0im                0.0249946+0.0769255im
            0.0625249+0.0im                0.0505837+0.0367512im
           -0.0483328+0.0im  ...    0.0483328+4.88679e-13im
            0.0373621+0.0im                0.0302265-0.0219609im
           -0.0288815+0.0im               0.00892487-0.0274679im
                    ⋮          ⋱
          -2.07638e-7+0.0im      6.41637e-8+1.97476e-7im
           1.60508e-7+0.0im      1.29854e-7+9.43441e-8im
          -1.24075e-7+0.0im  ...   1.24075e-7+8.21851e-17im
           9.59122e-8+0.0im      7.75946e-8-5.63758e-8im
          -7.41417e-8+0.0im       2.29111e-8-7.0513e-8im
           5.73128e-8+0.0im     -1.77106e-8-5.45077e-8im
          -4.43037e-8+0.0im     -3.58425e-8-2.60411e-8im
           3.42475e-8+0.0im  ...   -3.42475e-8-3.63968e-18im
          -2.64739e-8+0.0im      -2.14178e-8+1.5561e-8im
           2.04648e-8+0.0im     -6.32396e-9+1.94632e-8im
          -1.58196e-8+0.0im      4.88853e-9+1.50453e-8im
           1.22288e-8+0.0im      9.89333e-9+7.18792e-9im        )
```

```
In [36]: cond(Y)
```

```
Out[36]: 5.187627048504662 5e7
```

```
In [37]: norm(inv(Y)*ΔJ*Y), cond(Y)*norm(ΔJ)
```

```
Out[37]: (0.7730166686343102,0.7730166686333213)
```

2.5.5 Example - Normal matrix

```
In [38]: using SpecialMatrices
```

```
In [39]: n=6
         C=Circulant(rand(n))
```

```
Out[39]: 6x6 SpecialMatrices.Circulant{Float64}:
          0.280027   0.456382   0.89517    0.147186   0.314493   0.0329194
          0.0329194  0.280027   0.456382   0.89517    0.147186   0.314493
          0.314493   0.0329194  0.280027   0.456382   0.89517    0.147186
          0.147186   0.314493   0.0329194  0.280027   0.456382   0.89517
          0.89517    0.147186   0.314493   0.0329194  0.280027   0.456382
          0.456382   0.89517    0.147186   0.314493   0.0329194  0.280027
```

```
In [40]: λ=eigvals(full(C))

Out[40]: 6-element Array{Complex{Float64},1}:
            2.12618+0.0im
           0.853202+0.0im
          -0.227341+0.86961im
          -0.227341-0.86961im
          -0.422269+0.136152im
          -0.422269-0.136152im

In [41]: ΔC=rand(n,n)*0.0001

Out[41]: 6x6 Array{Float64,2}:
          2.70169e-5  4.43039e-5  4.50529e-5  7.88329e-5  4.25021e-5  7.55726e-5
          6.86612e-5  5.53859e-5  8.13803e-5  2.71019e-5  6.88746e-5  6.87128e-5
          1.1089e-5   3.69796e-5  3.66697e-5  4.31724e-5  6.25283e-5  3.94477e-5
          1.01807e-5  6.73907e-5  1.49378e-6  9.07601e-5  7.39689e-5  6.037e-5
          3.62829e-5  7.80455e-5  3.79155e-5  2.09202e-5  1.5202e-5   1.46986e-5
          9.25536e-5  2.45492e-5  9.49463e-5  2.76453e-5  3.174e-5    5.73445e-5

In [42]: @show norm(ΔC)
         μ=eigvals(C+ΔC)

norm(ΔC) = 0.00029795097944865007


Out[42]: 6-element Array{Complex{Float64},1}:
            2.12647+0.0im
           0.853175+0.0im
          -0.227349+0.869626im
          -0.227349-0.869626im
          -0.422252+0.136135im
          -0.422252-0.136135im
```

2.5.6 Example - Hermitian matrix

```
In [43]: m=10
         n=6
         A=rand(m,n)
         # Some scaling
         D=diagm((rand(n)-0.5)*exp(20))
         A=A*D

Out[43]: 10x6 Array{Float64,2}:
          -1.40598e7  -1.53282e7  4.81769e6  5.6818e7   -2.22987e7  -1.35568e8
          -3.30994e6  -4.32384e6  1.78038e6  1.0496e8   -1.8647e7   -1.38361e8
          -3.30849e7  -2.04635e7  6.17044e5  6.31594e7  -7.19669e7  -9.56312e6
          -1.0634e8   -3.14615e7  1.68204e6  5.7415e7   -8.16104e7  -1.27169e7
          -4.09174e7  -1.07478e7  8.57308e5  1.03076e8  -4.40167e7  -1.33705e8
          -6.53127e7  -2.94391e6  2.0912e6   1.0963e8   -8.96735e7  -9.95375e7
          -1.22917e8  -3.90819e6  3.76049e6  1.49196e8  -6.12102e6  -4.72304e7
```

```
 -1.36662e8  -5.28349e5  1.71862e6  8.64603e7  -9.99323e7  -1.10229e8
 -3.41082e6  -2.45257e7  4.90168e6  1.25894e8  -1.19024e8  -1.67825e8
 -1.22193e8  -2.80482e7  1.20269e6  2.79709e7  -7.63784e7  -1.2892e8
```

```
In [44]: A=cor(A)
```

```
Out[44]: 6x6 Array{Float64,2}:
  1.0         -0.0505982    0.28378       0.148572  0.133623   -0.397334
 -0.0505982    1.0          0.00824342    0.589239  0.353699   -0.178353
  0.28378      0.00824342   1.0           0.384523  0.10674    -0.366137
  0.148572     0.589239     0.384523      1.0       0.17392    -0.137651
  0.133623     0.353699     0.10674       0.17392   1.0         0.0695847
 -0.397334    -0.178353    -0.366137     -0.137651  0.0695847   1.0
```

```
In [45]: ΔA=cor(rand(m,n)*D)*1e-5
```

```
Out[45]: 6x6 Array{Float64,2}:
  1.0e-5       -1.75051e-6   7.27812e-7  -2.84144e-6   2.61253e-6  -1.4968e-6
 -1.75051e-6    1.0e-5      -5.44976e-6   6.24717e-6  -1.08082e-6   4.28063e-6
  7.27812e-7   -5.44976e-6   1.0e-5      -3.11945e-6  -4.48206e-6   9.84006e-7
 -2.84144e-6    6.24717e-6  -3.11945e-6   1.0e-5      -3.09465e-6   4.83014e-6
  2.61253e-6   -1.08082e-6  -4.48206e-6  -3.09465e-6   1.0e-5      -7.72464e-7
 -1.4968e-6     4.28063e-6   9.84006e-7   4.83014e-6  -7.72464e-7   1.0e-5
```

```
In [46]: B=A+ΔA
```

```
Out[46]: 6x6 Array{Float64,2}:
  1.00001    -0.0506       0.283781     0.148569  0.133626   -0.397336
 -0.0506      1.00001      0.00823797   0.589246  0.353698   -0.178349
  0.283781    0.00823797   1.00001      0.38452   0.106736   -0.366136
  0.148569    0.589246     0.38452      1.00001   0.173917   -0.137646
  0.133626    0.353698     0.106736     0.173917  1.00001     0.0695839
 -0.397336   -0.178349    -0.366136    -0.137646  0.0695839   1.00001
```

```
In [47]: λ,U=eig(A)
         μ=eigvals(B)
         [λ μ]
```

```
Out[47]: 6x2 Array{Float64,2}:
 0.198374  0.19838
 0.580301  0.580309
 0.768204  0.768214
 0.918354  0.918368
 1.44494   1.44495
 2.08983   2.08984
```

```
In [48]: # Residual bounds - how close is μ, y to λ[2],X[:,2]
         k=3
         μ=round(λ[k],2)
         y=round(U[:,k],2)
         y=y/norm(y)
```

```
Out[48]: 6-element Array{Float64,1}:
           0.229279
           0.378809
          -0.687837
          -0.239248
          -0.0697805
          -0.51837

In [49]: μ

Out[49]: 0.77

In [50]: # Fact 9
         r=A*y-μ*y

Out[50]: 6-element Array{Float64,1}:
          -0.000531079
          -0.00334765
           0.000334566
          -0.00302493
          -0.00252885
           0.00203196

In [51]: minimum(abs(μ-λ)), norm(r)

Out[51]: (0.0017962451153789027,0.005592394255532198)

In [52]: # Fact 10 - μ is Rayleigh quotient
         μ=(y'*A*y)[] # Vector, unfortunately
         r=A*y-μ*y

Out[52]: 6-element Array{Float64,1}:
          -0.000124534
          -0.00267597
          -0.000885071
          -0.00344915
          -0.00265258
           0.00111282

In [53]: η=min(abs(μ-λ[k-1]),abs(μ-λ[k+1]))

Out[53]: 0.15012666998626978

In [54]: μ-λ[k], norm(r)^2/η

Out[54]: (2.309646400422416e-5,0.00018738054558551144)

In [55]: # Eigenvector bound
         # cos(θ)
         cosθ=dot(y,U[:,k])
         # sin(θ)
         sinθ=sqrt(1-cosθ^2)
         sinθ,norm(r)/η
```

Out[55]: (0.005544671937376312,0.035329161020332664)

In [56]:
```
# Residual bounds - Fact 13
U=eigvecs(A)
Q=round(U[:,1:3],2)
# Orthogonalize
X,R=qr(Q)
M=X'*A*X
R=A*X-X*M
μ=eigvals(M)
λ, μ, R
```

Out[56]:
```
([0.19837443442964567,0.5803008298044705,0.7682037548846211,0.9183535213348951,1.44
6x3 Array{Float64,2}:
  0.0018205    0.00128987   -0.000503279
  0.00580417  -0.00253157   -0.00327183
  0.00408486   0.000309214  -0.00117756
  0.00657818  -0.00195991   -0.00303458
  0.00191     -0.00100452   -0.00231384
 -0.00366677  -0.000649976   0.000661031)
```

In [57]:
```
# The entries of μ are not ordered - which algorithm was called?
issym(M)
```

Out[57]: false

In [58]:
```
M=Hermitian(M)
R=A*X-X*M
μ=eigvals(M)
```

Out[58]:
```
3-element Array{Float64,1}:
 0.198439
 0.580316
 0.768229
```

In [59]:
```
η=λ[4]-λ[3]
```

Out[59]: 0.150149766450274

In [60]:
```
norm(λ[1:3]-μ), vecnorm(R)^2/η
```

Out[60]: (7.140567499891654e-5,0.0010312354761840806)

In [61]:
```
# Fact 13
M=A[1:3,1:3]
H=A[4:6,4:6]
E=A[4:6,1:3]
# Block-diagonal matrix
B=cat([1,2],M,H)
```

```
Out[61]: 6x6 Array{Float64,2}:
           1.0        -0.0505982   0.28378     0.0         0.0         0.0
          -0.0505982   1.0         0.00824342  0.0         0.0         0.0
           0.28378     0.00824342  1.0         0.0         0.0         0.0
           0.0         0.0         0.0         1.0         0.17392    -0.137651
           0.0         0.0         0.0         0.17392     1.0         0.0695847
           0.0         0.0         0.0        -0.137651    0.0695847   1.0
```

```
In [62]: η=minimum(abs(eigvals(M)-eigvals(H)))
         μ=eigvals(B)
         [λ μ], 2*norm(E)^2/(η+sqrt(η^2+4*norm(E)^2))
```

```
Out[62]: (
         6x2 Array{Float64,2}:
          0.198374  0.710213
          0.580301  0.741316
          0.768204  1.00285
          0.918354  1.0673
          1.44494   1.19139
          2.08983   1.28694 ,

         0.9191335682117601)
```

```
In [63]: # Eigenspace bounds - Fact 14
         B=A+ΔA
         μ,V=eig(B)
```

```
Out[63]: ([0.19837978573850565,0.5803093118649132,0.7682141929345702,0.9183679137342919,1.44
         6x6 Array{Float64,2}:
           0.218703  -0.606799   -0.230288    0.450177   0.459302   0.342511
           0.59387    0.0550786  -0.383002   -0.201056  -0.523811   0.427543
           0.362037   0.308178    0.687362   -0.072474   0.324656   0.436847
          -0.536699  -0.439542    0.237067   -0.336018  -0.269445   0.526353
          -0.259439   0.330311    0.0676698   0.773572  -0.375119   0.282631
           0.336249  -0.481129    0.516751    0.200856  -0.442659  -0.390038)
```

```
In [64]: # sin(Θ(U[:,1:3],V[:,1:3]))
         X=U[:,1:3]
         Q=V[:,1:3]
         cosθ=svdvals(sqrtm(Q'*Q)*Q'*X*sqrtm(X'*X))
         sinθ=sqrt(1-cosθ.^2)
```

```
Out[64]: 3-element Array{Float64,1}:
          3.78149e-7
          7.37222e-6
          2.08993e-5
```

```
In [65]: # Bound
         M=Q'*A*Q
```

```
Out[65]: 3x3 Array{Float64,2}:
           0.198374   2.0401e-6   -2.8022e-6
```

```
 2.0401e-6  0.580301     4.77924e-7
-2.8022e-6  4.77924e-7   0.768204
```

In [66]: R=A*Q-Q*M

Out[66]: 6x3 Array{Float64,2}:

```
 8.10106e-9  -2.37492e-6   9.01564e-7
 4.25945e-8   4.66762e-6   1.19302e-6
-4.76213e-7  -2.10329e-7  -2.17054e-7
-3.30246e-7   3.88803e-6   4.05916e-7
 1.01383e-6   1.28973e-6   3.54605e-6
 6.87361e-7   7.28328e-7   9.24148e-7
```

In [67]: eigvals(M), λ

Out[67]: ([0.1983744344317503,0.768203754952121,0.5803008298595141],[0.19837443442964567,0.5

In [68]: η=abs(eigvals(M)[3]-λ[4])
vecnorm(sinθ), vecnorm(R)/η

Out[68]: (2.2164694638019082e-5,2.3385305497259946e-5)

In [ ]:

3 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 I. Slapničar, Symmetric Matrix Eigenvalue Techniques and the references therein.

3.1 Prerequisites

The reader should be familiar with basic linear algebra concepts and facts on eigenvalue decomposition and perturbation theory

3.2 Competences

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

3.3 Backward error and stability

3.3.1 Definitions

If the value of a function $f(x)$ is computed with an algorithm 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.

3.4 Basic methods

3.4.1 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

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

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:

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

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

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

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

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

3.4.2 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 $x_k \to U_{:i}$. 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 $(A - \mu I)v_{k+1} = x_k$ for v_{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.

3.4.3 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::Float64)
            y=A*x
            ν=x⋅y
            steps=1
            while norm(y-ν*x)>tol
                x=y/norm(y)
                y=A*x
                ν=x⋅y
                steps+=1
            end
            ν, y/norm(y), steps
        end
```

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

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

```
Out[2]: 6x6 Array{Int64,2}:
          6   6   7  -9   8   3
          6  -7   1  -8  -9  -1
          7   1  -2   8   1   6
         -9  -8   8   3  -5   0
          8  -9   1  -5  -4   1
          3  -1   6   0   1  -3
```

```
In [3]: x0=rand(-9:9,n)
```

```
Out[3]: 6-element Array{Int64,1}:
          3
         -4
```

```
 -7
  1
  4
  7
```

In [4]:
```
ν,x=myPower(A,x0,1e-10)
```

Out[4]:
```
(-20.49531327253087,[0.16758272352745,-0.6876969003212962,0.19160389254062582,-0.356
```

In [5]:
```
eigvals(A)
```

Out[5]:
```
6-element Array{Float64,1}:
 -20.4953
 -14.4238
  -5.10669
   3.20668
  10.4406
  19.3785
```

In [6]:
```
eigvecs(A)[:,1]
```

Out[6]:
```
6-element Array{Float64,1}:
  0.167583
 -0.687697
  0.191604
 -0.356511
 -0.570036
 -0.101172
```

In [7]:
```
ν-eigvals(A)[1]
```

Out[7]:
```
7.105427357601002e-15
```

In [8]:
```
# Deflation
function myDeflation(A::Array,x::Vector)
    n,m=size(A)
    # Need to convert x to 2D array
    X,R=qr(x[:,:],thin=false)
    full(Symmetric(X[:,2:n]'*A*X[:,2:n]))
end
```

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

In [9]:
```
A1=myDeflation(A,x)
```

Out[9]:
```
5x5 Array{Float64,2}:
  9.25944   1.57741   -6.70387     6.26016    2.63911
  1.57741  -3.58393   10.2867      0.981951   5.73749
 -6.70387  10.2867    -0.0258976  -3.00149    0.837181
  6.26016   0.981951  -3.00149    10.1269     4.27873
  2.63911   5.73749    0.837181    4.27873   -2.28116
```

```
In [10]: eig(A1)

Out[10]: ([-14.423840475157174,-5.106686937787888,3.206681076605001,10.440644533549182,19.37
         5x5 Array{Float64,2}:
          -0.186553    -0.242036    0.679319    0.0712869   0.66337
           0.734096    -0.0897466   0.265209   -0.617835   -0.0314934
          -0.598052    -0.404004    0.0837317  -0.598785   -0.336987
          -0.00927876  -0.213442   -0.67282    -0.301501    0.640911
          -0.261817     0.851227    0.092151   -0.404662    0.186068 )

In [11]: myPower(A1,rand(n-1),1e-10)

Out[11]: (19.378515075321836,[-0.663370312568391,0.031493436455098,0.33698667179172537,-0.64

In [12]: # Put it all together - eigenvectors are ommited for the sake of simplicty
         function myPowerMethod(A::Array, tol::Float64)
             n,m=size(A)
             λ=Array(Float64,n)
             for i=1:n
                 λ[i],x,steps=myPower(A,rand(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}:
          -20.4953
           19.3785
          -14.4238
           10.4406
           -5.10669
            3.20668

In [14]: # QR iteration
         function myQRIteration(A::Array, tol::Float64)
             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]: myQRIteration(A,1e-5)
```

```
Out[15]: (
6x6 Array{Float64,2}:
 -20.4953          9.68275e-6     -5.66664e-16   ...  -7.26264e-16  -1.93106e-15
   9.68275e-6     19.3785          2.1609e-15         -1.87188e-15   6.66685e-15
   3.01868e-44     1.27563e-37   -14.4238              2.87903e-15  -2.64321e-15
  -3.10098e-86    -1.58142e-79     2.66892e-41         1.34905e-15  -3.8093e-16
  -3.85406e-179   -1.28199e-172   -2.98173e-134       -5.10669      -2.11824e-15
  -1.7799e-238    -6.2409e-232    -1.94293e-193  ...  -5.11907e-59   3.20668    ,

299)
```

3.5 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, $X^TAX = T$.
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.

3.5.1 Facts

1. Tridiagonal form is not unique.
2. 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 H be the **Householder reflector**,
$$v = a + \operatorname{sign}(a_1)\|a\|_2 e_1, \quad H = I - 2\frac{vv^T}{v^Tv},$$
and set
$$H_1 = \begin{bmatrix} 1 & \\ & H \end{bmatrix}.$$
Then
$$H_1AH_1 = \begin{bmatrix} \alpha & a^TH \\ Ha & HBH \end{bmatrix} = \begin{bmatrix} \alpha & \nu e_1^T \\ \nu e_1 & A_1 \end{bmatrix}, \quad \nu = -\operatorname{sign}(a_1)\|a\|_2.$$
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 $T = X^TAX$, $X = H_1H_2\cdots H_{n-2}$.
3. H does not depend on the normalization of v. With the normalization $v_1 = 1$, $a_{2:n-1}$ can be overwritten by $v_{2:n-1}$ (and v_1 does not need to be stored).
4. The matrix 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.

5. 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.

6. If the matrix 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 $2n^3$.

7. 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.

8. Tridiagonalization using Givens rotations requires $(n-1)(n-2)/2$ 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.

9. 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 $(n-1, n)$, $(n-2, n-1)$, ..., $(2, 3)$. 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.

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

11. 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. *Julia wappers for those routines do nost exist yet!*

```
In [16]: function myTridiag{T}(A::Array{T})
             # Normalized Householder vectors are stored in the lower triangular part of A
             # below the first subdiagonal
             n,m=size(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(Float64)
        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[16]: myTridiag (generic function with 1 method)

In [17]:
```
T,H=myTridiag(map(Float64,A))
```

Out[17]:
```
(
6x6 SymTridiagonal{Float64}:
   6.0      -15.4596     0.0       0.0       0.0      0.0
 -15.4596    -0.912134  -6.41264   0.0       0.0      0.0
   0.0       -6.41264   -3.46285  -10.3522   0.0      0.0
   0.0        0.0      -10.3522    -2.76773 -10.4707  0.0
   0.0        0.0        0.0      -10.4707   -8.74382 2.63392
   0.0        0.0        0.0        0.0       2.63392 2.88653,

6x6 Array{Float64,2}:
  0.0        0.0         0.0        0.0       0.0  0.0
  0.0        0.0         0.0        0.0       0.0  0.0
  0.326194   0.0         0.0        0.0       0.0  0.0
 -0.419392   0.593872    0.0        0.0       0.0  0.0
  0.372793   0.0850364   0.104828   0.0       0.0  0.0
  0.139797  -0.310585   -0.428859   0.793906  0.0  0.0)
```

In [18]:
```
eigvals(A), eigvals(T)
```

Out[18]: ([-20.495313272530876,-14.423840475157226,-5.106686937787903,3.2066810766049962,10

In [19]:
```
# Extract X
function myTridiagX{T}(H::Array{T})
    n,m=size(H)
    X = eye(T,n)
    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[19]: myTridiagX (generic function with 1 method)

In [20]: X=myTridiagX(H)

Out[20]: 6x6 Array{Float64,2}:
          1.0   0.0        0.0        0.0         0.0        0.0
          0.0  -0.388108  -0.328103  -0.520291   -0.121776   0.675417
          0.0  -0.452792  -0.480295   0.706421    0.237732   0.0935362
          0.0   0.582162  -0.677942   0.0647512  -0.443475  -0.0248869
          0.0  -0.517477  -0.239093  -0.294915   -0.317696  -0.697959
          0.0  -0.194054   0.380649   0.37296    -0.794389   0.217478

In [21]: # Fact 7: norm(ΔX)<ϕ*eps()
         X'*X

Out[21]: 6x6 Array{Float64,2}:
          1.0   0.0           0.0           0.0           0.0           0.0
          0.0   1.0          -4.16334e-17   6.93889e-17  -2.77556e-17   4.16334e-17
          0.0  -4.16334e-17   1.0           1.94289e-16   1.66533e-16  -2.77556e-17
          0.0   6.93889e-17   1.94289e-16   1.0          -1.11022e-16  -4.16334e-17
          0.0  -2.77556e-17   1.66533e-16  -1.11022e-16   1.0           0.0
          0.0   4.16334e-17  -2.77556e-17  -4.16334e-17   0.0           1.0

In [22]: X'*A*X

Out[22]: 6x6 Array{Float64,2}:
            6.0          -15.4596       -4.44089e-16  …    4.44089e-16  -6.66134e-16
          -15.4596        -0.912134     -6.41264           0.0           6.10623e-16
           -4.44089e-16   -6.41264      -3.46285           2.22045e-15  -2.66454e-15
           -4.44089e-16    7.77156e-16 -10.3522          -10.4707        1.44329e-15
            4.44089e-16    4.44089e-16   1.9984e-15       -8.74382       2.63392
           -6.66134e-16    8.32667e-16  -1.11022e-15  …    2.63392       2.88653

In [23]: # Tridiagonalization using Givens rotations
         function myTridiagG{T}(A::Array{T})
             n,m=size(A)
             X=eye(T,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[23]: myTridiagG (generic function with 1 method)

In [24]: methods(givens)
```

```
Out[24]: # 3 methods for generic function "givens":
         givens{T}(A::AbstractArray{T,2}, i1::Integer, i2::Integer, col::Integer) at linalg/
         givens{T}(f::T, g::T, i1::Integer, i2::Integer) at linalg/givens.jl:237
         givens{T}(f::T, g::T, i1::Integer, i2::Integer, cols::Integer) at deprecated.jl:49

In [25]: Tg,Xg=myTridiagG(map(Float64,A))

Out[25]: (
         6x6 SymTridiagonal{Float64}:
           6.0      15.4596      0.0        0.0       0.0       0.0
          15.4596  -0.912134    6.41264    0.0       0.0       0.0
           0.0       6.41264   -3.46285  -10.3522    0.0       0.0
           0.0       0.0      -10.3522    -2.76773  10.4707    0.0
           0.0       0.0        0.0       10.4707   -8.74382  -2.63392
           0.0       0.0        0.0        0.0      -2.63392   2.88653,

         6x6 Array{Float64,2}:
          1.0   0.0        0.0        0.0        0.0        0.0
          0.0   0.388108  -0.328103  -0.520291   0.121776   0.675417
          0.0   0.452792  -0.480295   0.706421  -0.237732   0.0935362
          0.0  -0.582162  -0.677942   0.0647512  0.443475  -0.0248869
          0.0   0.517477  -0.239093  -0.294915   0.317696  -0.697959
          0.0   0.194054   0.380649   0.37296    0.794389   0.217478 )

In [26]: T

Out[26]: 6x6 SymTridiagonal{Float64}:
            6.0     -15.4596      0.0        0.0        0.0      0.0
          -15.4596   -0.912134   -6.41264    0.0        0.0      0.0
            0.0      -6.41264    -3.46285  -10.3522     0.0      0.0
            0.0       0.0       -10.3522    -2.76773  -10.4707   0.0
            0.0       0.0         0.0      -10.4707    -8.74382  2.63392
            0.0       0.0         0.0        0.0        2.63392  2.88653

In [27]: Xg'*Xg

Out[27]: 6x6 Array{Float64,2}:
          1.0   0.0           0.0           0.0          0.0           0.0
          0.0   1.0          -5.55112e-17   2.77556e-17  2.77556e-17  -2.77556e-17
          0.0  -5.55112e-17   1.0          -5.55112e-17  5.55112e-17   2.77556e-17
          0.0   2.77556e-17  -5.55112e-17   1.0          1.11022e-16   1.11022e-16
          0.0   2.77556e-17   5.55112e-17   1.11022e-16  1.0           8.32667e-17
          0.0  -2.77556e-17   2.77556e-17   1.11022e-16  8.32667e-17   1.0

In [28]: Xg'*A*Xg

Out[28]: 6x6 Array{Float64,2}:
           6.0          15.4596         0.0          …  -8.88178e-16   1.22125e-15
          15.4596       -0.912134       6.41264        -6.66134e-16  -9.99201e-16
           0.0           6.41264       -3.46285         4.44089e-16   5.55112e-16
           4.44089e-16   1.22125e-15  -10.3522         10.4707        1.66533e-15
          -8.88178e-16  -1.11022e-15    1.11022e-15    -8.74382      -2.63392
           1.22125e-15  -3.88578e-16    4.44089e-16  …  -2.63392       2.88653
```

3.6 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 $T - \mu I$.

3.6.1 Definition

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

3.6.2 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 $T_{n-1,n} \to 0$. 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 iteration on s 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 `myEigQR()` 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 `myEigQR()` 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.

3.6.3 Examples

```
In [29]: function myTridEigQR{T}(A1::SymTridiagonal{T})
             A=deepcopy(A1)
             n=length(A.dv)
             λ=Array(T,n)
             Temp=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
                 Temp=A[1:2,1:2]
                 G,r=givens(Temp-μ*I,1,2,1)
                 Temp=(G*Temp)*G'
                 return diag(Temp)[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
            Temp=A[1:3,1:3]
            G,r=givens(Temp-μ*I,1,2,1)
            Temp=(G*Temp)*G'
            A.dv[1:2]=diag(Temp)[1:2]
            A.ev[1:2]=diag(Temp,-1)
            bulge=Temp[3,1]
            # Bulge chasing
            for i = 2 : n-2
                Temp=A[i-1:i+2,i-1:i+2]
                Temp[3,1]=bulge
                Temp[1,3]=bulge
                G,r=givens(Temp,2,3,1)
                Temp=(G*Temp)*G'
                A.dv[i:i+1]=diag(Temp)[2:3]
                A.ev[i-1:i+1]=diag(Temp,-1)
                bulge=Temp[4,2]
            end
            # Last rotation
            Temp=A[n-2:n,n-2:n]
            Temp[3,1]=bulge
            Temp[1,3]=bulge
            G,r=givens(Temp,2,3,1)
            Temp=(G*Temp)*G'
            A.dv[n-1:n]=diag(Temp)[2:3]
            A.ev[n-2:n-1]=diag(Temp,-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[29]: myTridEigQR (generic function with 1 method)
```

```
In [30]: ?findfirst
```

```
search: findfirst
```

```
Out[30]:
```

```
findfirst(A,v)
```

Return the index of the first element equal to `v` in `A`.

```
findfirst(A)
```

Return the index of the first non-zero value in `A` (determined by `A[i]!=0`).

```
findfirst(predicate, A)
```

Return the index of the first element of `A` for which `predicate` returns `true`.

```
In [31]: λ=eigvals(T)

Out[31]: 6-element Array{Float64,1}:
          -20.4953
          -14.4238
           -5.10669
            3.20668
           10.4406
           19.3785

In [32]: λ1=myTridEigQR(T)

Out[32]: 6-element Array{Float64,1}:
           19.3785
          -20.4953
          -14.4238
           -5.10669
           10.4406
            3.20668

In [33]: (sort(λ)-sort(λ1))./sort(λ)

Out[33]: 6-element Array{Float64,1}:
           1.73343e-16
          -3.69463e-16
          -6.95698e-16
          -5.53955e-16
           0.0
          -9.16663e-16
```

3.6.4 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 [34]: U=LAPACK.stein!(T.dv,T.ev,λ)

Out[34]: 6x6 Array{Float64,2}:
           0.167583    0.513938   -0.328833   0.0719933   0.258877    0.726259
           0.28721     0.678967   -0.236244   0.0130081  -0.0743602  -0.628493
           0.473084    0.19161     0.638223  -0.181917   -0.492456    0.237787
           0.60045    -0.217706    0.247684   0.109144    0.707451   -0.13534
           0.548867   -0.431793   -0.575672   0.117582   -0.405535    0.0511561
          -0.0618287   0.0657008   0.189695   0.967376   -0.141399    0.00817008

In [35]: # Orthogonality
         U'*U
```

```
Out[35]: 6x6 Array{Float64,2}:
           1.0           1.25767e-16   4.68375e-17  …   5.20417e-18  -7.66531e-17
           1.25767e-16   1.0          -1.61329e-16      1.56125e-17  -6.59195e-17
           4.68375e-17  -1.61329e-16   1.0             -3.46945e-17   5.42101e-17
          -1.38778e-17   0.0           0.0              2.77556e-17  -1.38778e-17
           5.20417e-18   1.56125e-17  -3.46945e-17      1.0           2.19009e-17
          -7.66531e-17  -6.59195e-17   5.42101e-17  …   2.19009e-17   1.0
```

```
In [36]: # Residual
         T*U-U*diagm(λ)
```

```
Out[36]: 6x6 Array{Float64,2}:
           0.0          8.88178e-16   8.88178e-16  …  -8.88178e-16   7.10543e-15
           1.77636e-15  1.77636e-15   0.0              1.11022e-16   0.0
           1.77636e-15  0.0           0.0              8.88178e-16   2.66454e-15
           3.55271e-15  0.0           0.0             -1.77636e-15  -4.44089e-16
           0.0          0.0          -4.44089e-16      8.88178e-16   3.33067e-16
          -4.44089e-16  1.11022e-16  -1.11022e-16  …   2.22045e-16   5.55112e-17
```

```
In [37]: # Some timings - n=100, 200, 400 myTridEigQR() is 200x slower!?
         n=400
         Tbig=SymTridiagonal(rand(n),rand(n-1))
         @time myTridEigQR(Tbig);
         @time λbig=eigvals(Tbig);
         @time LAPACK.stein!(Tbig.dv,Tbig.ev,λbig);

  0.413223 seconds (4.01 M allocations: 257.388 MB, 9.86% gc time)
  0.004167 seconds (13 allocations: 13.156 KB)
  0.014493 seconds (31 allocations: 1.266 MB)
```

```
In [38]: 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);

  0.143462 seconds (14 allocations: 63.109 KB)
  0.410030 seconds (38 allocations: 30.722 MB, 0.66% gc time)
  0.705899 seconds (221.61 k allocations: 72.390 MB, 0.65% gc time)
```

```
In [39]: @which eigvals(Tbig)
```

```
Out[39]: eigvals{T}(A::SymTridiagonal{T}) at linalg/tridiag.jl:129
```

Alternatively, the rotations in `myTridEigQR()` can be accumulated to compute the eignevctors. This is not optimal, but is instructive. We make use of Julia's multiple dispatch feature.

```
In [40]: function myTridEigQR{T}(A1::SymTridiagonal{T},U::Array{T})
             # U is either the identity matrix or the output from myTridiagX()
```

```
A=deepcopy(A1)
n=length(A.dv)
λ=Array(T,n)
Temp=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
    Temp=A[1:2,1:2]
    G,r=givens(Temp-μ*I,1,2,1)
    Temp=(G*Temp)*G'
    U*=G'
    return diag(Temp)[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
    Temp=A[1:3,1:3]
    G,r=givens(Temp-μ*I,1,2,1)
    Temp=(G*Temp)*G'
    U[:,1:3]*=G'
    A.dv[1:2]=diag(Temp)[1:2]
    A.ev[1:2]=diag(Temp,-1)
    bulge=Temp[3,1]
    # Bulge chasing
    for i = 2 : n-2
        Temp=A[i-1:i+2,i-1:i+2]
        Temp[3,1]=bulge
        Temp[1,3]=bulge
        G,r=givens(Temp,2,3,1)
        Temp=(G*Temp)*G'
        U[:,i-1:i+2]=U[:,i-1:i+2]*G'
        A.dv[i:i+1]=diag(Temp)[2:3]
        A.ev[i-1:i+1]=diag(Temp,-1)
        bulge=Temp[4,2]
    end
    # Last rotation
    Temp=A[n-2:n,n-2:n]
    Temp[3,1]=bulge
    Temp[1,3]=bulge
    G,r=givens(Temp,2,3,1)
    Temp=(G*Temp)*G'
    U[:,n-2:n]*=G'
    A.dv[n-1:n]=diag(Temp)[2:3]
```

```
        A.ev[n-2:n-1]=diag(Temp,-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], 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[:
    λ, U
end
```

Out[40]: `myTridEigQR (generic function with 2 methods)`

In [41]:
```
λ,U=myTridEigQR(T,eye(T))
```

Out[41]:
```
([19.378515075321854,-20.49531327253087,-14.423840475157228,-5.1066869377879085,10
6x6 Array{Float64,2}:
 -0.726259     0.167583   -0.513938    0.328833   0.258877    0.0719933
  0.628493     0.28721    -0.678967    0.236244  -0.0743602   0.0130081
 -0.237787     0.473084   -0.19161    -0.638223  -0.492456   -0.181917
  0.13534      0.60045     0.217706   -0.247684   0.707451    0.109144
 -0.0511561    0.548867    0.431793    0.575672  -0.405535    0.117582
 -0.00817008  -0.0618287  -0.0657008  -0.189695  -0.141399    0.967376 )
```

In [42]:
```
# Orthogonality
U'*U
```

Out[42]:
```
6x6 Array{Float64,2}:
  1.0           2.24972e-16  -1.81821e-16  ...  6.61363e-17  -6.93889e-18
  2.24972e-16  1.0            2.62811e-16      1.21431e-17   1.38778e-17
 -1.81821e-16  2.62811e-16   1.0               6.59195e-17  -4.16334e-17
 -4.83554e-17  1.33574e-16   9.36751e-17      1.17961e-16  -2.77556e-17
  6.61363e-17  1.21431e-17   6.59195e-17      1.0            2.77556e-17
 -6.93889e-18  1.38778e-17  -4.16334e-17  ...  2.77556e-17   1.0
```

In [43]:
```
# EVD
U'*(T*U)
```

Out[43]:
```
6x6 Array{Float64,2}:
 19.3785         -6.93889e-16    7.01609e-15  ...    9.99201e-16  -1.97758e-16
  1.85615e-16  -20.4953          5.89806e-16        4.02456e-15   4.16334e-16
  4.34548e-15   -5.55112e-16  -14.4238             -1.13798e-15   0.0
  1.54043e-15   -3.35842e-15   -2.88658e-15        -4.996e-16    -1.11022e-16
  7.35523e-16    8.60423e-16   -1.4988e-15         10.4406         1.66533e-16
 -3.88578e-16    4.44089e-16    0.0           ...    0.0            3.20668
```

3.6.5 Symmetric QR method

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

In [44]:
```
function mySymEigQR{T}(A::Array{T})
    Tr,H=myTridiag(A)
```

```
    X=myTridiagX(H)
    # λ, U
    myTridEigQR(Tr,X)
end
```

Out[44]: mySymEigQR (generic function with 1 method)

In [45]:
```
λ,U=mySymEigQR(map(Float64,A))
```

Out[45]:
```
([19.378515075321854,-20.49531327253087,-14.423840475157228,-5.1066869377879085,10
6x6 Array{Float64,2}:
 -0.726259   0.167583  -0.513938   0.328833    0.258877   0.0719933
 -0.235609  -0.687697   0.116152   0.0483563  -0.223763   0.636916
 -0.087688   0.191604   0.649758   0.143709    0.660318   0.277023
  0.558744  -0.356511  -0.441126   0.303597    0.519739   0.0617492
 -0.286337  -0.570036   0.241635   0.0528993   0.175112  -0.707969
 -0.123137  -0.101172  -0.217284  -0.879721    0.38223    0.0859127)
```

In [46]:
```
U'*U
```

Out[46]:
```
6x6 Array{Float64,2}:
  1.0           5.89806e-17  -2.42861e-17  ...   9.02056e-17   2.94903e-17
  5.89806e-17   1.0           2.81025e-16      -1.249e-16    -1.42247e-16
 -2.42861e-17   2.81025e-16   1.0             -2.22045e-16  -4.51028e-17
 -1.38778e-17  -2.77556e-16  -8.32667e-17      1.11022e-16   0.0
  9.02056e-17  -1.249e-16    -2.22045e-16      1.0           4.16334e-17
  2.94903e-17  -1.42247e-16  -4.51028e-17  ...   4.16334e-17   1.0
```

In [47]:
```
U'*A*U
```

Out[47]:
```
6x6 Array{Float64,2}:
 19.3785          9.99201e-16    2.77556e-15  ...   2.22045e-15   9.99201e-16
  1.33227e-15   -20.4953        -3.4972e-15      -3.33067e-16   8.04912e-16
  2.60902e-15    -3.83027e-15  -14.4238          -2.44249e-15   6.10623e-16
 -2.22045e-16     5.55112e-17   -3.66374e-15     -3.33067e-15  -4.44089e-16
  1.94289e-15    -1.83187e-15   -2.55351e-15     10.4406         7.21645e-16
  3.26128e-16     2.04003e-15    8.32667e-16  ...   7.21645e-16   3.20668
```

In [ ]:

4 Symmetric Eigenvalue Decomposition - Algorithms for Tridiagonal Matrices

---

Due to their importance, there is plethora of excellent algorithms for symmetric tridiagonal matrices.
For more details, see I. Slapničar, Symmetric Matrix Eigenvalue Techniques and the references therein.

4.1 Prerequisites

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

4.2 Competences

The reader should be able to apply adequate algorithm to a given symmetric tridiagonal matrix, and to assess its speed and the accuracy of the solution.

4.3 Bisection and inverse iteration

The bisection method is convenient if only part of the spectrum is needed. If the eigenvectors are needed, as well, they can be efficiently computed by the inverse iteration method.

4.3.1 Facts

A is a real symmetric $n\times n$ matrix and T is a real symmetric tridiagonal $n\times n$ matrix.

1. *(Application of Sylvester's Theorem)* Let $\alpha,\beta\in\mathbb{R}$ with $\alpha<\beta$. The number of eigenvalues of A in the interval $[\alpha,\beta)$ is equal to $\nu(A-\beta I)-\nu(A-\alpha I)$. By systematically choosing the intervals $[\alpha,\beta)$, the bisection method pinpoints each eigenvalue of A to any desired accuracy.

2. The factorization $T-\mu I=LDL^T$, where $D=\operatorname{diag}(d_1,\ldots,d_n)$ and L is the unit lower bidiagonal matrix, is computed as:

$$\begin{aligned} &d_1=T_{11}-\mu,\quad d_i=(T_{ii}-\mu)-\frac{T_{i,i-1}^2}{d_{i-1}},\quad i=2,\ldots n,\\ &L_{i+1,i}=\frac{T_{i+1,i}}{d_i},\quad i=1,\ldots,n-1. \end{aligned}$$

   Since the matrices T and D have the same inertia, this recursion enables an efficient implementation of the bisection method for T.

3. The factorization from Fact 2 is essentially Gaussian elimination without pivoting. Nevertheless, if $d_i\neq 0$ for all i, the above recursion is very stable. Even when $d_{i-1}=0$ for some i, if the IEEE arithmetic is used, the computation will continue and the inertia will be computed correctly. Namely, in that case, we would have $d_i=-\infty$, $l_{i+1,i}=0$, and $d_{i+1}=t_{i+1.i+1}-\mu$.

4. Computing one eigenvalue of T by using the recursion from Fact 2 and bisection requires $O(n)$ operations. The corresponding eigenvector is computed by inverse iteration. The convergence is very fast, so the cost of computing each eigenvector is also $O(n)$ operations. Therefore, the overall cost for computing all eigenvalues and eigenvectors is $O(n^2)$ operations.

5. Both, bisection and inverse iteration are highly parallel since each eigenvalue and eigenvector can be computed independently.

6. If some of the eigenvalues are too close, the corresponding eigenvectors computed by inverse iteration may not be sufficiently orthogonal. In this case, it is necessary to orthogonalize these eigenvectors (for example, by the modified Gram–Schmidt procedure). If the number of close eigenvalues is too large, the overall operation count can increase to $O(n^3)$.

7. The EVD computed by bisection and inverse iteration satisfies the error bounds from previous notebook.

8. The bisection method for tridiagonal matrices is implemented in the LAPACK subroutine DSTEBZ. This routine can compute all eigenvalues in a given interval or the eigenvalues from λ_l to λ_k, where $l < k$, and the eigenvalues are ordered from smallest to largest. Inverse iteration (with reorthogonalization) is implemented in the LAPACK subroutine DSTEIN.

In [1]:
```
n=6
T=SymTridiagonal(rand(n),rand(n-1))
```

Out[1]:
```
6x6 SymTridiagonal{Float64}:
 0.990185   0.0391408  0.0        0.0        0.0        0.0
 0.0391408  0.114456   0.691343   0.0        0.0        0.0
 0.0        0.691343   0.0822071  0.848197   0.0        0.0
 0.0        0.0        0.848197   0.954516   0.0233555  0.0
 0.0        0.0        0.0        0.0233555  0.392447   0.593533
 0.0        0.0        0.0        0.0        0.593533   0.551464
```

In [2]:
```
λ,U=eig(T)
```

Out[2]:
```
([-0.828635006860597,-0.1270871282014696,0.40487417757166044,0.991877277042248,1.07
6x6 Array{Float64,2}:
  0.0118844   -0.000568245  -0.0526644    0.998353    -0.00704987  0.0180584
 -0.552251     0.0162206     0.787543     0.0431558   -0.0144874   0.269182
  0.752677    -0.00563501    0.333811    -0.00175099  -0.0196376   0.567122
 -0.358142    -0.0118305    -0.514919    -0.037053    -0.0110756   0.77779
  0.00866055   0.752522     -0.00497852   0.00431712   0.658119    0.0217153
 -0.0037246   -0.658236      0.0201578    0.00581805   0.752404    0.0126093)
```

In [3]:
```
methods(LAPACK.stebz!)
```

Out[3]:
```
# 2 methods for generic function "stebz!":
stebz!(range::Char, order::Char, vl::Float64, vu::Float64, il::Integer, iu::Integer
stebz!(range::Char, order::Char, vl::Float32, vu::Float32, il::Integer, iu::Integer
```

In [4]:
```
λ1, rest=LAPACK.stebz!('A','E',1.0,1.0,1,1,2*eps(),T.dv,T.ev)
```

```
Out[4]: ([-0.8286350068606007,-0.1270871282014669,0.4048741775716592,0.9918772770422479,1.07
```

```
In [5]: λ-λ1
```

```
Out[5]: 6-element Array{Float64,1}:
          3.66374e-15
         -2.72005e-15
          1.22125e-15
          1.11022e-16
         -6.66134e-16
          0.0
```

```
In [6]: U1=LAPACK.stein!(T.dv,T.ev,λ1)
```

```
Out[6]: 6x6 Array{Float64,2}:
          0.0118844   -0.000568245  -0.0526644    0.998353    -0.00704987  0.0180584
         -0.552251     0.0162206     0.787543     0.0431558   -0.0144874   0.269182
          0.752677    -0.00563501    0.333811    -0.00175099  -0.0196376   0.567122
         -0.358142    -0.0118305    -0.514919    -0.037053    -0.0110756   0.77779
          0.00866055   0.752522     -0.00497852   0.00431712   0.658119    0.0217153
         -0.0037246   -0.658236      0.0201578    0.00581805   0.752404    0.0126093
```

```
In [7]: # Let us compute just some eigenvalues - from 2nd to 4th
        λ2,rest=LAPACK.stebz!('I','E',1.0,1.0,2,4,2*eps(),T.dv,T.ev)
```

```
Out[7]: ([-0.1270871282014669,0.4048741775716592,0.9918772770422479],[1,1,1],[6])
```

```
In [8]: U2=LAPACK.stein!(T.dv,T.ev,λ2)
```

```
Out[8]: 6x3 Array{Float64,2}:
         -0.000568245  -0.0526644    0.998353
          0.0162206     0.787543     0.0431558
         -0.00563501    0.333811    -0.00175099
         -0.0118305    -0.514919    -0.037053
          0.752522     -0.00497852   0.00431712
         -0.658236      0.0201578    0.00581805
```

4.4 Divide-and-conquer

This is currently the fastest method for computing the EVD of a real symmetric tridiagonal matrix T. It is based on splitting the given tridiagonal matrix into two matrices, then computing the EVDs of the smaller matrices and computing the final EVD from the two EVDs.
T is a real symmetric tridiagonal matrix of order n and $T = U\Lambda U^T$ is its EVD.

4.4.1 Facts

1. Let T be partitioned as

$$T = \begin{bmatrix} T_1 & \alpha_k e_k e_1^T \\ \alpha_k e_1 e_k^T & T_2 \end{bmatrix}.$$

   We assume that T is unreduced, that is, $\alpha_i \neq 0$ for all i. Further, we assume that $\alpha_i > 0$ for all i, which can be easily be attained by diagonal similarity with a diagonal matrix of signs. Let

$$\hat{T}_1 = T_1 - \alpha_k e_k e_k^T, \qquad \hat{T}_2 = T_2 - \alpha_k e_1 e_1^T.$$

In other words, $\hat{T}_1$ is equal to T_1 except that T_{kk} is replaced by $T_{kk} - \alpha_k$, and $\hat{T}_2$ is equal to T_2 except that $T_{k+1,k+1}$ is replaced by $T_{k+1,k+1} - \alpha_k$. Let $\hat{T}_i = \hat{U}_i \hat{\Lambda}_i \hat{U}_i^T$, $i = 1, 2$, be the respective EVDs and let $v = \begin{bmatrix} \hat{U}_1^T e_k \\ \hat{U}_2^T e_1 \end{bmatrix}$ (v consists of the last column of $\hat{U}_1^T$ and the first column of $\hat{U}_2^T$). Set $\hat{U} = \hat{U}_1 \oplus \hat{U}_2$ and $\hat{\Lambda} = \hat{\Lambda}_1 \oplus \hat{\Lambda}_2$. Then

$$T = \begin{bmatrix} \hat{U}_1 & \\ & \hat{U}_2 \end{bmatrix} \left[\begin{bmatrix} \hat{\Lambda}_1 & \\ & \hat{\Lambda}_2 \end{bmatrix} + \alpha_k v v^T \right] \begin{bmatrix} \hat{U}_1^T & \\ & \hat{U}_2^T \end{bmatrix} = \hat{U} (\hat{\Lambda} + \alpha_k v v^T) \hat{U}^T.$$

If $\hat{\Lambda} + \alpha_k v v^T = X \Lambda X^T$ is the EVD of the rank-one modification of the diagonal matrix $\hat{\Lambda}$, then $T = U \Lambda U^T$, where $U = \hat{U} X$ is the EVD of T. Thus, the original tridiagonal eigenvalue problem is reduced to two smaller tridiagonal eigenvalue problems and one eigenvalue problem for the diagonal-plus-rank-one matrix.

2. If all $\hat{\lambda}_i$ are different, then the eigenvalues λ_i of $\hat{\Lambda} + \alpha_k v v^T$ are solutions of the so-called secular equation,

$$1 + e_k \sum_{i=1}^{n} \frac{v_i^2}{\hat{\lambda}_i - \lambda} = 0.$$

The eigenvalues can be computed by bisection, or by some faster zero finder of the Newton type, and they need to be computed as accurately as possible. The corresponding eigenvectors are

$$x_i = (\hat{\Lambda} - \lambda_i I)^{-1} v.$$

3. Each λ_i and x_i in $O(n)$ operations, respectively, so the overall computational cost for computing the EVD of $\hat{\Lambda} + \alpha_k v v^T$ is $O(n^2)$ operations.

4. The method can be implemented so that the accuracy of the computed EVD is given by the bound from the previous notebook.

5. Tridiagonal Divide-and-conquer method is implemented in the LAPACK subroutine DSTEDC. This routine can compute just the eigenvalues or both, eigenvalues and eigenvectors.

The file lapack.jl contains wrappers for a selection of LAPACK routines needed in the current Julia `Base`. However, *all* LAPACK routines are in the compiled library, so additional wrappers can be easily written. Notice that arrays are passed directly and scalars as passed as pointers. The wrapper for `DSTEDC`, similar to the ones from the file `lapack.jl` follows.

In [9]:
```
# Part of the preamble of lapack.jl
const liblapack = Base.liblapack_name
import Base.blasfunc
# import ..LinAlg: BlasFloat, Char, BlasInt, LAPACKException,
    # DimensionMismatch, SingularException, PosDefException, chkstride1, chksquare
import Base.LinAlg.BlasInt
macro assertargsok() #Handle only negative info codes - use only if positive info c
    # is useful!
    :(info[1]<0 && throw(ArgumentError("invalid argument #$(-info[1]) to LAPACK cal
end
macro lapackerror() #Handle all nonzero info codes
    :(info[1]>0 ? throw(LAPACKException(info[1])) : @assertargsok )
    end
```

```
In [10]: for (stedc, elty) in
             ((:dstedc_,:Float64),
             (:sstedc_,:Float32))
             @eval begin
                 """
                 COMPZ is CHARACTER*1
                   = 'N':  Compute eigenvalues only.
                   = 'I':  Compute eigenvectors of tridiagonal matrix also.
                   = 'V':  Compute eigenvectors of original dense symmetric
                           matrix also.  On entry, Z contains the orthogonal
                           matrix used to reduce the original matrix to
                           tridiagonal form.
                 """
                 function stedc!(compz::Char, dv::Vector{$elty}, ev::Vector{$elty}, Z::Array
                     n = length(dv)
                     ldz=n
                     if length(ev) != n - 1
                         throw(DimensionMismatch("ev has length $(length(ev)) but needs one
                     end
                     w = deepcopy(dv)
                     u = deepcopy(ev)
                     lwork=5*n^2
                     work = Array($elty, lwork)
                     liwork=6+6*n+5*n*round(Int,ceil(log(n)/log(2)))
                     iwork = Array(BlasInt,liwork)
                     info = Array(BlasInt,1)
                     ccall(($(blasfunc(stedc)), liblapack), Void,
                         (Ptr{UInt8}, Ptr{BlasInt}, Ptr{$elty},
                         Ptr{$elty}, Ptr{$elty}, Ptr{BlasInt}, Ptr{$elty}, Ptr{BlasInt},
                         Ptr{BlasInt}, Ptr{BlasInt}, Ptr{BlasInt}),
                         &compz, &n, w,
                         u, Z, &ldz, work, &lwork,
                         iwork, &liwork, info)
                         @lapackerror
                     w,Z
                 end
             end
         end

In [11]: μ,Q=stedc!('I',T.dv,T.ev,eye(n))

Out[11]: ([-0.8286350068606008,-0.1270871282014671,0.40487417757165917,0.991877277042248,1.0
         6x6 Array{Float64,2}:
           0.0118844    0.000568245   0.0526644    0.998353     -0.00704987  -0.0180584
          -0.552251    -0.0162206    -0.787543     0.0431558    -0.0144874   -0.269182
           0.752677     0.00563501   -0.333811    -0.00175099   -0.0196376   -0.567122
          -0.358142     0.0118305     0.514919    -0.037053     -0.0110756   -0.77779
           0.00866055  -0.752522      0.00497852   0.00431712    0.658119    -0.0217153
          -0.0037246    0.658236     -0.0201578    0.00581805    0.752404    -0.0126093)

In [12]: λ-μ
```

```
Out[12]: 6-element Array{Float64,1}:
           3.77476e-15
          -2.52576e-15
           1.27676e-15
           0.0
          -8.88178e-16
           0.0
```

```
In [13]: Q'*(T*Q)
```

```
Out[13]: 6x6 Array{Float64,2}:
          -0.828635     -1.84314e-18  -4.00057e-16  ...  -5.20417e-18  -1.66018e-16
           0.0          -0.127087     -9.36751e-17        5.55112e-16  -1.00614e-16
          -3.14934e-16  -9.5193e-17    0.404874          -5.20417e-17   7.15573e-18
          -2.05385e-16  -3.3014e-17   -7.86385e-17       -4.77049e-17  -2.42211e-16
          -5.63785e-18   4.996e-16    -5.0307e-17         1.07062       3.46945e-18
          -2.76668e-16  -1.04083e-16  -4.70273e-17  ...   1.04083e-17   1.57363
```

```
In [14]: # Timings
         n=3000
         Tbig=SymTridiagonal(rand(n),rand(n-1));
```

```
In [15]: @time eig(Tbig);
         @time stedc!('I',Tbig.dv,Tbig.ev,eye(n));

  1.156802 seconds (225 allocations: 138.143 MB, 0.52% gc time)
  0.283952 seconds (24 allocations: 413.545 MB, 14.07% gc time)
```

4.5 MRRR

The method of Multiple Relatively Robust Representations

The computation of the tridiagonal EVD which satisfies the error stabdard error bounds such that the eigenvectors are orthogonal to working precision, all in $O(n^2)$ operations, has been the *holy grail* of numerical linear algebra for a long time. The method of Multiple Relatively Robust Representations does the job, except in some exceptional cases. The key idea is to implement inverse iteration more carefully. The practical algorithm is quite elaborate and the reader is advised to consider references.

The MRRR method is implemented in the LAPACK subroutine DSTEGR. This routine can compute just the eigenvalues, or both eigenvalues and eigenvectors.

```
In [16]: methods(LAPACK.stegr!)
```

```
Out[16]: # 3 methods for generic function "stegr!":
         stegr!(jobz::Char, range::Char, dv::Array{Float64,1}, ev::Array{Float64,1}, vl::Rea
         stegr!(jobz::Char, range::Char, dv::Array{Float32,1}, ev::Array{Float32,1}, vl::Rea
         stegr!(jobz::Char, dv::Array{T,1}, ev::Array{T,1}) at linalg/lapack.jl:3268
```

```
In [17]: LAPACK.stegr!('V',T.dv,T.ev)
```

```
Out[17]: ([-0.828635006860597,-0.1270871282014696,0.40487417757166044,0.991877277042248,1.07
         6x6 Array{Float64,2}:
           0.0118844   -0.000568245  -0.0526644    0.998353    -0.00704987  0.0180584
          -0.552251     0.0162206     0.787543     0.0431558   -0.0144874   0.269182
           0.752677    -0.00563501    0.333811    -0.00175099  -0.0196376   0.567122
          -0.358142    -0.0118305    -0.514919    -0.037053    -0.0110756   0.77779
           0.00866055   0.752522     -0.00497852   0.00431712   0.658119    0.0217153
          -0.0037246   -0.658236      0.0201578    0.00581805   0.752404    0.0126093)
```

```
In [18]: # Timings
         @time LAPACK.stegr!('V',Tbig.dv,Tbig.ev);
```

```
  1.190483 seconds (71 allocations: 138.087 MB, 2.61% gc time)
```

```
In [ ]:
```

5 Symmetric Eigenvalue Decomposition - Jacobi Method and High Relative Accuracy

---

The Jacobi method is the oldest method for EVD computations, dating back from 1864. The method does not require tridiagonalization. Instead, the method computes a sequence of orthogonally similar matrices which converge to a diagonal matrix of eigenvalues. In each step a simple plane rotation which sets one off-diagonal element to zero is performed.

For positive definite matrices, the method computes eigenvalues with high relative accuracy.

For more details, see I. Slapničar, Symmetric Matrix Eigenvalue Techniques and Z. Drmač, Computing Eigenvalues and Singular Values to High Relative Accuracy and the references therein.

5.1 Prerequisites

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

5.2 Competences

The reader should be able to recognise matrices which warrant high relative accuracy and to apply Jacobi method to them.

5.3 Jacobi method

A is a real symmetric matrix of order n and $A=U\Lambda U^T$ is its EVD.

5.3.1 Definitions

The **Jacobi method** forms a sequence of matrices,

$$A_0=A, \qquad A_{k+1}=G(i_k,j_k,c,s)A_kG(i_k,j_k,c,s)^T, \qquad k=1,2,\ldots,$$

where $G(i_k,j_k,c,s)$ is the orthogonal **plane rotation matrix**. The parameters c and s are chosen such that $[A_{k+1}]_{i_k j_k}=[A_{k+1}]_{j_k i_k}=0$.

The plane rotation is also called the **Jacobi rotation**.

The **off-norm** of A is

$$off(A)=\big(\sum_i\sum_{j\neq i}a_{ij}^2\big)^{1/2},$$

that is, off-norm is the Frobenius norm of the matrix consisting of all off-diagonal elements of A.

The choice of **pivot elements** $[A_k]_{i_kj_k}$ is called the **pivoting strategy**.

The **optimal pivoting strategy**, originally used by Jacobi, chooses pivoting elements such that $|[A_k]_{i_k j_k}|=\max_{i<j}|[A_k]_{ij}|$.

The **row-cyclic** pivoting strategy chooses pivot elements in the systematic row-wise order,

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4),\ldots,(2,n),(3,4),\ldots,(n-1,n).$$

Similarly, the column-cyclic strategy chooses pivot elements column-wise.

One pass through all matrix elements is called **cycle** or **sweep**.

5.3.2 Facts

1. The Jacobi rotations parameters c and s are computed as follows: If $[A_k]_{i_k j_k} = 0$, then $c = 1$ and $s = 0$, otherwise

$$\tau = \frac{[A_k]_{i_k i_k} - [A_k]_{j_k j_k}}{2[A_k]_{i_k j_k}}, \qquad t = \frac{\operatorname{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}}, \qquad c = \frac{1}{\sqrt{1+t^2}}, \qquad s = c \cdot t.$$

2. After each rotation, the off-norm decreases,

$$off^2(A_{k+1}) = off^2(A_k) - 2[A_k]^2_{i_k j_k}.$$

   With the appropriate pivoting strategy, the method converges in the sense that

$$off(A_k) \to 0, \qquad A_k \to \Lambda, \qquad \prod_{k=1}^{\infty} G(i_k, j_k, c, s)^T \to U.$$

3. For the optimal pivoting strategy the square of the pivot element is greater than the average squared element, $[A_k]^2_{i_k j_k} \geq off^2(A) \frac{1}{n(n-1)}$. Thus,

$$off^2(A_{k+1}) \leq \left(1 - \frac{2}{n(n-1)}\right) off^2(A_k)$$

   and the method converges.

4. For the row cyclic and the column cyclic pivoting strategies, the method converges. The convergence is ultimately **quadratic** in the sense that $off(A_{k+n(n-1)/2}) \leq \; const \cdot off^2(A_k)$, provided $off(A_k)$ is sufficiently small.

5. The EVD computed by the Jacobi method satisfies the standard error bounds.

6. The Jacobi method is suitable for parallel computation. There exist convergent parallel strategies which enable simultaneous execution of several rotations.

7. The Jacobi method is simple, but it is slower than the methods based on tridiagonalization. It is conjectured that standard implementations require $O(n^3 \log n)$ operations. More precisely, each cycle clearly requires $O(n^3)$ operations and it is conjectured that $\log n$ cycles are needed until convergence.

8. If A is positive definite, the method can be modified such that it reaches the speed of the methods based on tridiagonalization and at the same time computes the EVD with high relative accuracy.

5.3.3 Examples

```
In [1]: function myJacobi{T}(A::Array{T})
            n,m=size(A)
            U=eye(T,n)
            # Tolerance for rotation
            tol=sqrt(n)*eps(T)
            # Counters
            p=n*(n-1)/2
            sweep=0
            pcurrent=0
            # First criterion is for standard accuracy, second one is for relative accuracy
```

```
        # while sweep<30 && vecnorm(A-diagm(diag(A)))>tol
        while sweep<30 && pcurrent<p
            sweep+=1
            # Row-cyclic strategy
            for i = 1 : n-1
                for j = i+1 : n
                    # Check the tolerance - the first criterion is standard,
                    # the second one is for relative accuracy for PD matrices
                    # if A[i,j]!=zero(T)
                    if abs(A[i,j])>tol*sqrt(abs(A[i,i]*A[j,j]))
                        # Compute c and s
                        τ=(A[i,i]-A[j,j])/(2*A[i,j])
                        t=sign(τ)/(abs(τ)+sqrt(1+τ^2))
                        c=1/sqrt(1+t^2)
                        s=c*t
                        G=LinAlg.Givens(i,j,c,s)
                        A=G*A
                        # @show
                        A*=G'
                        A[i,j]=zero(T)
                        A[j,i]=zero(T)
                        U*=G'
                        pcurrent=0
                    else
                        pcurrent+=1
                    end
                end
            end
        end
        # λ, U
        # @show A
        diag(A), U
    end
```

Out[1]: myJacobi (generic function with 1 method)

In [2]:
```
n=4
A=full(Symmetric(rand(n,n)))
```

Out[2]:
```
4x4 Array{Float64,2}:
 0.859825  0.625873  0.245435  0.561921
 0.625873  0.97313   0.854558  0.413555
 0.245435  0.854558  0.377692  0.696606
 0.561921  0.413555  0.696606  0.386355
```

In [3]:
```
λ,U=myJacobi(A)
```

Out[3]:
```
([-0.5110259907281365,2.3931528143541243,0.5038727589045998,0.2110026480898031],
4x4 Array{Float64,2}:
  0.264145  0.479409  -0.821547   -0.159544
 -0.366291  0.613623   0.357112   -0.601472
  0.706437  0.464375   0.43904     0.304213
 -0.544981  0.421888  -0.0691035   0.72127 )
```

```
In [4]: U'*U

Out[4]: 4x4 Array{Float64,2}:
          1.0            0.0            -5.55112e-17  -3.33067e-16
          0.0            1.0             1.2837e-16   -1.11022e-16
         -5.55112e-17    1.2837e-16      1.0           1.04083e-16
         -3.33067e-16   -1.11022e-16     1.04083e-16   1.0
```

```
In [5]: A*U-U*diagm(λ)

Out[5]: 4x4 Array{Float64,2}:
          1.94289e-16  -4.44089e-16   5.55112e-16   4.16334e-17
          8.32667e-17  -6.66134e-16   5.55112e-17   0.0
          1.11022e-16  -4.44089e-16  -5.55112e-17   8.32667e-17
         -2.77556e-16  -4.44089e-16   1.66533e-16  -5.55112e-17
```

```
In [6]: # Positive definite matrix
        n=100
        A=rand(n,n)
        A=full(Symmetric(A'*A));
```

```
In [7]: λ,U=myJacobi(A)
        norm(U'*U-I),norm(A*U-U*diagm(λ))

Out[7]: (2.918223906221685e-14,3.9587120290006577e-11)
```

```
In [8]: λ

Out[8]: 100-element Array{Float64,1}:
         2532.18
            0.0131241
           31.1896
            0.00189852
           29.8623
            0.147798
           28.4812
            0.00333202
           27.7636
            0.0919647
            0.341706
           24.3592
            3.83947
            ⋮
            2.13628
            2.06048
            5.04817
            4.37197
            5.36254
            5.7869
            4.02991
            3.21938
```

```
4.51509
4.29691
3.33341
2.84398
```

```
In [9]: cond(A)

Out[9]: 1.3337682453386835e6

In [10]: # Now the standard QR method
         λ,U=eig(A);

In [11]: norm(U'*U-I),norm(A*U-U*diagm(λ))

Out[11]: (3.050836346283519e-13,3.0408891376813924e-12)
```

5.4 Relative perturbation theory

A is a real symmetric PD matrix of order n and $A = U\Lambda U^T$ is its EVD.

5.4.1 Definition

The **scaled matrix** of the matrix A is the matrix

$$A_S = D^{-1}AD^{-1}, \quad D = \operatorname{diag}(\sqrt{A_{11}}, \sqrt{A_{22}}, \ldots, \sqrt{A_{nn}}).$$

5.4.2 Facts

1. The above diagonal scaling is nearly optimal: $\kappa_2(A_S) \leq n \min_{D=\text{diag}} \kappa(DHD) \leq n\kappa_2(H)$.

2. Let A and $\tilde{A} = A + \Delta A$ both be positive definite, and let their eigenvalues have the same ordering. Then
$$\frac{|\lambda_i - \tilde{\lambda}_i|}{\lambda_i} \leq \frac{\|D^{-1}(\Delta A)D^{-1}\|_2}{\lambda_{\min}(A_S)} \equiv \|A_S^{-1}\|_2 \|\Delta A_S\|_2.$$
If λ_i and $\tilde{\lambda}_i$ are simple,
$$\|U_{:,i} - \tilde{U}_{:,i}\|_2 \leq \frac{\|A_S^{-1}\|_2 \|\Delta A_S\|_2}{\min\limits_{j \neq i} \dfrac{|\lambda_i - \lambda_j|}{\sqrt{\lambda_i \lambda_j}}}.$$
These bounds are much sharper than the standard bounds for matrices for which $\kappa_2(A_S) \ll \kappa_2(A)$.

3. Jacobi method with the relative stopping criterion $|A_{ij}| \leq tol\sqrt{A_{ii}A_{jj}}$ for all $i \neq j$ and some user defined tolerance tol (usually $tol = n\varepsilon$), computes the EVD with small scaled backward error $\|\Delta A_S\| \leq \varepsilon\, O(\|A_S\|_2) \leq O(n)\varepsilon$, *provided* that $\kappa_2([A_k]_S)$ does not grow much during the iterations. There is overwhelming numerical evidence that the scaled condition does not grow much, and the growth can be monitored, as well.

5.4.3 Example - Scaled matrix

```
In [12]: n=10
         A=rand(n,n)
         A=full(Symmetric(A'*A));
         AS=map(Float64,[A[i,j]/sqrt(A[i,i]*A[j,j]) for i=1:n, j=1:n])

Out[12]: 10x10 Array{Float64,2}:
          1.0       0.775925  0.78287   0.860785  …  0.795853  0.88939   0.789678
          0.775925  1.0       0.809157  0.862699     0.879892  0.84235   0.713707
          0.78287   0.809157  1.0       0.735156     0.810281  0.766874  0.706796
          0.860785  0.862699  0.735156  1.0          0.721396  0.839212  0.800589
          0.83921   0.860412  0.869796  0.885945     0.853105  0.837846  0.774648
          0.673876  0.807851  0.499119  0.710431  …  0.754399  0.628388  0.463822
          0.5802    0.864702  0.641829  0.700768     0.739358  0.752179  0.731881
          0.795853  0.879892  0.810281  0.721396     1.0       0.895524  0.636262
          0.88939   0.84235   0.766874  0.839212     0.895524  1.0       0.863853
          0.789678  0.713707  0.706796  0.800589     0.636262  0.863853  1.0

In [13]: cond(AS)

Out[13]: 6438.469701303686

In [14]: # Strong scaling
         D=exp(50*(rand(n)-0.5))

Out[14]: 10-element Array{Float64,1}:
             1.3056e10
           187.247
             0.519645
          7480.92
             8.47713e6
             4.76788e6
             0.0105304
             9.31052e-9
             0.0179848
             4.95575e-5

In [15]: H=diagm(D)*AS*diagm(D)

Out[15]: 10x10 Array{Float64,2}:
           1.7046e20      1.89691e12  …      2.08837e8       5.10941e5
           1.89691e12  35061.4               2.83669         0.00662284
           5.31139e9      78.7326            0.00716696      1.82016e-5
           8.40739e13      1.20845e6       112.91            0.296807
           9.28818e16      1.36575e9         1.27737e5     325.434
           4.19486e16      7.21226e8  …  53883.8           109.594
           7.9769e7        1.70501           0.000142453     3.8194e-7
          96.7427          1.53397e-6        1.49953e-10     2.93575e-13
           2.08837e8       2.83669           0.000323451     7.69934e-7
           5.10941e5       0.00662284        7.69934e-7      2.45594e-9

In [16]: cond(H)
```

```
Out[16]: 5.681132633167512 6e38

In [17]: λ,U=myJacobi(H)

Out[17]: ([1.7046028869315386e20,5758.217122299141,0.019613240938357607,9.29216761000908 5e6
         10x10 Array{Float64,2}:
          1.0           5.2848e-10    -1.478e-11    ...  -1.27742e-12   -4.84392e-16
          1.11282e-8    0.999947      -0.00211354        -0.000122616    2.52239e-7
          3.11591e-11   0.00211793     0.99933            0.0318588     -3.62716e-5
          4.93217e-7   -0.0100759      3.99904e-5         1.43881e-6    -4.29729e-9
          0.000544889  -6.35136e-6    -5.16255e-8        -1.92148e-9     1.84727e-12
          0.00024609   -1.31079e-5     6.34499e-8   ...   2.954e-9       3.21631e-13
          4.67963e-13   5.58004e-5    -0.0205358          0.084339      -0.003132
          5.67538e-19   3.68398e-11   -5.2104e-9          4.35468e-7    -7.23864e-5
          1.22514e-12   5.06493e-5    -0.0302288          0.995926      -0.0013797
          2.99742e-15   6.91997e-8    -6.97774e-5         0.0016394      0.999994    )

In [18]: λ1,U1=eig(H)

Out[18]: ([1.7046028869315386e20,2.516659278642747e13,8.494442473043522e12,9.29216761000890 7
         10x10 Array{Float64,2}:
          -1.0            0.000595889  -4.8785e-5    ...  -4.84256e-16    1.53703e-19
          -1.11282e-8    -1.64435e-5   -7.25104e-6         2.52265e-7    -4.98721e-12
          -3.11591e-11   -3.12262e-8    6.07341e-8        -3.6278e-5     -5.51285e-9
          -4.93217e-7    -0.000450076   0.000112905       -4.2976e-9      4.85689e-13
          -0.000544889   -0.874739      0.484594           1.84764e-12   -2.60639e-16
          -0.00024609    -0.484594     -0.874739     ...   3.21098e-13   -6.02047e-16
          -4.67963e-13   -1.09884e-9   -9.63041e-11       -0.0031321      5.80933e-8
          -5.67538e-19   -6.94531e-16  -1.63013e-16       -7.23861e-5     1.0
          -1.22514e-12   -5.32631e-10   5.38972e-10       -0.00137978    -5.61164e-7
          -2.99742e-15   -1.32373e-12   4.34529e-12        0.999994       7.23883e-5 )

In [19]: [sort(λ) sort(λ1)]

Out[19]: 10x2 Array{Float64,2}:
             3.00045e-19     3.00057e-19
             3.21207e-10     3.21203e-10
             1.55128e-5      1.55121e-5
             2.41645e-5      2.41626e-5
             0.0196132       0.0196122
          5758.22         5758.22
             9.29217e6       9.29217e6
             8.49444e12      8.49444e12
             2.51666e13      2.51666e13
             1.7046e20       1.7046e20

In [20]: # Check with BigFloat
         λ2,U2=myJacobi(map(BigFloat,H))

Out[20]: (BigFloat[1.704602886931538122856351185871238430256551219373736947824283237015803 6
         10x10 Array{BigFloat,2}:
          9.99999821267888555774383271396600737237381216385192629149224936990454154886629 8e-
```

```
 1.112815090823393410531458308637509661235248724503474845905713611390092868664836e-
 3.115909627309810380779716391626884296927762505080774447935526063417266544813718e-
 4.932172114464236092082703502913159212129759904410521624035786555738928864493836e-
 5.448886474329050232869647377023619978333121071979227631723476390528209907905431e-
 2.460900433885298738788816736303036134701878598545976295621070938288962142582876e-
 4.679627151840760918928530268479756915021272595289877634864675650106655725282158e-
 5.675382862269266323957070779936467665849838129550208023199296451906421639945426e-
 1.225138301520120770501671069701115167900567536136729090038284675477412696089217e-
 2.997420861084002762432563185557037856081962673467160424129896322082368447404322e-
```

```
In [21]: # Relative error is eps()*cond(AS)
         (sort(λ2)-sort(λ))./sort(λ2)
```

```
Out[21]: 10-element Array{BigFloat,1}:
 -3.999179931145441112702375974089419511298710550913244690217470891489332018129192e
 -2.306100765461286095081901666002979972306439586915259834290159435133537534470686e
 -2.253284861014899976751857483263722176432996190091471233673661381397945937285884e
 -1.998970492048416430045027175766613016378993370181437895279101368694365541505427e
 -3.982669614836893846312651647180995759458308334260618798595021499950504334863851e
 -2.536198724229273532807567562620040349893096007761613519969131798997053420308456e
  5.244923428887690677137345100226672103928146045758583318276622090228122075604643e
 -3.496147003669947861156990725333293991158798653517648610859410537849789851807331e
 -1.040196208487708199373214190631126598117557991739115376316957831008931507892269e
 -2.654598629881955855775874571362378843782143927382395304004922771867755034629747e
```

5.5 Indefinite matrices

5.5.1 Definition

Spectral absolute value of the matrix A is the matrix is $|A|_{spr} = (A^2)^{1/2}$ (positive definite part of the polar decomposition of A).

5.5.2 Facts

1. The above perturbation bounds for positive definite matrices essentially hold with A_S replaced by $[|A|_{spr}]_S$.
2. Jacobi method can be modified to compute the EVD with small backward error $\|\Delta[|A|_{spr}]_S\|_2$.

The details of the indefinite case are beyond the scope of this course, and the reader should consider references.

```
In [ ]:
```

5.6 # 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 I. Slapničar, Symmetric Matrix Eigenvalue Techniques and the references therein.

5.7 Prerequisites

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

5.8 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.

5.9 Lanczos method

A is a real symmetric matrix of order n.

5.9.1 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.

5.9.2 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 $T = X^TAX$ 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, $\nu = T_{k+1,k}$, provides information about accuracy:

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

Further, there are k eigenvalues $\tilde\lambda_1,\ldots,\tilde\lambda_k$ of A such that $|\lambda_i-\tilde\lambda_i|\leq\nu$, and for the corresponding eigenvectors, we have

$$\sin 2\Theta(U_{:,i},\tilde U_{:,i})\leq \frac{2\nu}{\min_{j\neq i}|\lambda_i-\tilde\lambda_j|}.$$

5. In practical implementations, ν 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]` $\mathbf{z}=\mathbf{z}-t_{ii}X_{:,i}-t_{i,i-1}X_{:,i-1}$ is replaced by $\mathbf{z}=\mathbf{z}-\sum_{j=1}^{i-1}(\mathbf{z}^TX(:,j))X(:,j)$. 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.

5.9.3 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
        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]: ([-5.352288224253318,-4.2193320767911375,-3.1907174771879343,-1.4737942171177876,0.0
        100x10 Array{Float64,2}:
         -0.0525375   -0.112847     0.0303205   0.150346   …  -0.106634     0.0986278
          0.0151725   -0.255843    -0.0787192   0.0214709     -0.22782      0.0945524
          0.100657     0.086244    -0.0778236   0.0115238     -0.136879     0.103228
         -0.10192      0.0910078   -0.110662   -0.0276567      0.169202     0.105208
          0.105235    -0.0294468   -0.0890833   0.0176073      0.147307     0.099769
         -0.118116     0.0292497    0.117138   -0.0477277  …   0.14184      0.102818
         -0.161154     0.0419611    0.0962546  -0.0638417      0.0332455    0.0977051
         -0.0210168    0.173314    -0.0501723  -0.0304011      0.116159     0.105094
         -0.143728     0.0937599    0.019697    0.117466       0.0608433    0.0933777
          0.00385748  -0.067783    -0.154644   -0.0687833     -0.187444     0.0970876
          0.0851419    0.110459    -0.133847    0.0161762  …   0.00950674   0.0974904
         -0.090655    -0.120565    -0.0499511   0.0744572     -0.00181588   0.099804
          0.037298    -0.121657     0.165484    0.0638076     -0.116132     0.103317
          ⋮                                                ⋱
          0.0553882   -0.042741     0.0345078  -0.141037       0.0549658    0.106478
         -0.0980244   -0.00662564   0.0935468  -0.0346602      0.0331928    0.104762
```

```
  0.0528576   -0.132375    0.123439    0.0699764  …   0.0779792    0.105547
 -0.106006    -0.00673235  0.170498    0.0475436     -0.124974     0.100194
  0.0722739    0.0560896   0.0704082  -0.1046         0.0167597    0.107939
 -0.0374963    0.0240778  -0.0854657   0.0441158     -0.199124     0.101306
  0.0131762   -0.0301349   0.138985    0.112099       0.0877181    0.0907318
  0.0007123   -0.00956891 -0.100135    0.0126032  …  -0.000313653  0.101851
 -0.150348    -0.00384459  0.0969491   0.0457243      0.0524795    0.100278
 -0.0396394    0.00818025  0.0559044  -0.0402859     -0.103406     0.108299
 -0.042183     0.147695    0.191335   -0.06388       -0.16874      0.0936431
 -0.0908279   -0.0491071  -0.0114865   0.177517       0.0768843    0.113033 ,

100x10 Array{Float64,2}:
 0.0554417     0.114285    -0.140432     …   0.00961807  -0.171202
 0.0579906     0.105743    -0.225334        -0.068022    -0.0668504
 0.0611571     0.104891    -0.0133489       -0.105821    -0.00568206
 0.086221      0.0596919    0.035153         0.0845895    0.286896
 0.0795725     0.0665293   -0.0681834       -0.0251777   -0.00980104
 0.136564     -0.0424879    0.106513     …   0.0338113   -0.0984841
 0.0735298     0.051477     0.261911        -0.0419475   -0.0604064
 0.172646     -0.0996042    0.0429803       -0.0139458    0.0698767
 0.154959     -0.0903062    0.0246825        0.0821852   -0.144929
 0.0471032     0.121375    -0.0744087       -0.137179     0.0536672
 0.113582      0.00315446  -0.106804     …  -0.0579251    0.027967
 0.000501112   0.212918    -0.0749958        0.0504378    0.0423027
 0.136857     -0.0238376   -0.161593         0.189833     0.0242407
 ⋮                                       ⋱
 0.126114     -0.0138871    0.0688412        0.027259    -0.0720161
 0.108062      0.0146213    0.0938811        0.106977     0.00155092
 0.0445229     0.142419    -0.0525868    …   0.212575     0.0531409
 0.104384      0.0169799    0.0263259        0.0633625   -0.0643486
 0.160958     -0.0692524    0.000101742     -0.0634959   -0.0677984
 0.072694      0.0830557   -0.0602781       -0.0440815   -0.0122363
 0.0922272     0.0190697    0.0191022        0.150596    -0.121337
 0.117464      0.00357924  -0.0916713    …  -0.00381793  -0.0364474
 0.103506      0.017213     0.044306         0.157068     0.038814
 0.134201     -0.0218748    0.0338348       -0.0610711   -0.129825
 0.165831     -0.115479     0.121573         0.0987965    0.0447898
 0.0972863     0.0580987   -0.0107879        0.125041    -0.147334 ,

2.8768046111404932)
```

In [4]:
```
# Orthogonality
X'*X
```

Out[4]:
```
10x10 Array{Float64,2}:
  1.0          -3.39138e-16   6.2862e-16   …  -7.8583e-16   -1.52656e-16
 -3.39138e-16   1.0           2.3756e-15      -2.87444e-15   1.23686e-15
  6.2862e-16    2.3756e-15    1.0             -1.96891e-16   2.74216e-15
  2.61943e-16  -6.96491e-16  -2.55763e-15      3.01321e-15  -4.00721e-16
 -8.01442e-16  -2.67321e-15   1.79978e-16      6.55725e-16  -3.08781e-15
 -2.34188e-16   5.76796e-16   2.44856e-15  …  -2.9126e-15    7.71952e-16
  6.76542e-16   2.7465e-15    7.49292e-16      3.86843e-16   2.80678e-15
```

```
 1.29237e-16  -9.45424e-16  -2.41289e-15      2.88658e-15  -4.33681e-16
-7.8583e-16   -2.87444e-15  -1.96891e-16      1.0          -3.11556e-15
-1.52656e-16   1.23686e-15   2.74216e-15     -3.11556e-15   1.0
```

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

```
Out[5]: 10x10 Array{Float64,2}:
 38.3549       21.197         7.53217e-14  ...  -8.9373e-14    2.12053e-14
 21.197        10.9962        3.3095           -4.724e-14      1.99285e-14
  7.42947e-14   3.3095        0.657931         -4.78784e-16    4.70457e-15
 -3.26822e-15  -9.86364e-15   3.00848           2.3731e-15     9.71445e-17
 -8.56711e-14  -4.48075e-14   4.72712e-16      -7.76289e-17    1.35439e-15
  3.74006e-15   8.96505e-15   4.47819e-15  ...   5.80092e-15  -1.27676e-15
  8.52599e-14   4.84482e-14   1.10068e-15       5.6205e-16     2.42514e-15
 -1.55778e-14  -1.61936e-14  -4.07031e-15       2.65912       -2.94903e-16
 -9.03617e-14  -4.84786e-14  -7.43329e-16       0.129416       2.43372
  2.14108e-14   2.03995e-14   5.18823e-15       2.43372        0.613717
```

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

```
Out[6]: (2.8768046111404924,2.8768046111404932)
```

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

```
Out[7]: 10x10 Array{Float64,2}:
 -5.35229       6.8695e-16    3.29597e-15  ...  -4.6213e-15   -7.56339e-15
  6.69603e-16  -4.21933       2.77339e-16      -5.93275e-15    3.97044e-14
  3.27863e-15   2.62811e-16  -3.19072           3.78864e-15    1.35152e-14
  4.01415e-15   6.14959e-16  -2.03201e-15      -7.28584e-17   -1.50855e-13
 -2.74086e-15  -5.26922e-15  -6.70514e-15       8.00054e-15    6.07396e-14
 -1.00072e-16   4.53186e-15  -3.04421e-15  ...   4.96521e-15  -1.03143e-13
  4.73059e-15   2.09902e-15   1.09223e-15       7.97105e-15   -1.71564e-14
  7.35523e-16   1.80758e-15  -3.48419e-15       2.45602e-14    1.99771e-14
 -4.08007e-15  -6.49827e-15   4.35069e-15       5.46043        7.07767e-16
 -7.10543e-15   3.80251e-14   1.47382e-14       1.72085e-15   49.9545
```

```
In [8]: U'*U
```

```
Out[8]: 10x10 Array{Float64,2}:
  1.0          -3.64292e-17  -2.50234e-16  ...   1.50054e-16  -5.37764e-17
 -3.64292e-17   1.0           2.14889e-16      -4.33681e-18    6.40113e-16
 -2.50234e-16   2.14889e-16   1.0               5.56738e-16    1.46367e-16
 -8.29198e-16   1.23165e-16  -4.09395e-16       8.08381e-16   -2.88658e-15
  6.52256e-16   1.3288e-15    6.54858e-17       8.32667e-17    1.30451e-15
  7.54605e-17   8.10983e-17  -5.67038e-16  ...   1.4138e-15   -2.11983e-15
 -3.25749e-16   2.59558e-16  -8.14148e-16       1.67959e-15   -3.20599e-16
  8.50015e-17   3.11383e-16   1.28695e-16       3.22398e-15    3.89445e-16
  1.50054e-16  -4.33681e-18   5.56738e-16       1.0           -2.25514e-17
 -5.37764e-17   6.40113e-16   1.46367e-16      -2.25514e-17    1.0
```

```
In [9]: λeig,Ueig=eig(A)
```

```
Out[9]: ([-5.48003,-5.32076,-5.04457,-4.88439,-4.56518,-4.53665,-4.47582,-4.25972,-4.12503,-
        100x100 Array{Float64,2}:
          0.0592035    0.0891002   -0.180887    ...   0.134598    -0.0986278
          0.000670684 -0.0262451   -0.0479776        0.26648     -0.0945524
          0.0480105   -0.0911351    0.1256           0.0926254   -0.103228
         -0.0158513    0.102907     0.0612032       -0.208299    -0.105208
          0.0790446   -0.0940853    0.0406253       -0.0768202   -0.099769
         -0.0162617    0.123512    -0.121886    ...  -0.113737    -0.102818
         -0.212545     0.0693098    0.040899        -0.00423835  -0.0977051
         -0.00995656   0.0104302    0.103472        -0.181751    -0.105094
         -0.0800883    0.113806    -0.0554854       -0.0282255   -0.0933777
          0.0268181    0.00923486  -0.0355246        0.227432    -0.0970876
         -0.0169435   -0.116483    -0.13201     ...   0.0228273   -0.0974904
          0.0683804    0.148453     0.061575        -0.0176286   -0.099804
         -0.149464    -0.122749    -0.0950543        0.0962935   -0.103317
          ⋮                                     ⋱
         -0.0389816   -0.106494    -0.128093        -0.00313634  -0.106478
         -0.089783     0.0512802   -0.126291         0.0378943   -0.104762
         -0.022375    -0.0771865    0.0794228   ...  -0.0901734   -0.105547
         -0.125872     0.0588563    0.0608166        0.0663129   -0.100194
          0.0489479   -0.0645721   -0.0814932        0.103618    -0.107939
         -0.00048028   0.0392894    0.124091         0.190371    -0.101306
          0.0407354    0.0203691   -0.174934        -0.0951056   -0.0907318
          0.105109     0.0729856    0.101161    ...  -0.0449658   -0.101851
          0.0583337    0.223372     0.0578846       -0.05914     -0.100278
          0.0105502    0.073987     0.122019         0.156876    -0.108299
          0.0892431    0.106993    -0.00171867       0.0366752   -0.0936431
         -0.174505    -0.0014195    0.0605028        0.0142365   -0.113033 )
```

```
In [10]: ?eigs
```

```
search: eigs eigvecs eigvals eigvals! leading_ones leading_zeros
```

```
Out[10]:
```

```
..  eigs(A, B; nev=6, ncv=max(20,2*nev+1), which="LM", tol=0.0, maxiter=300, sigma=nothing,

Computes generalized eigenvalues ``d`` of ``A`` and ``B`` using Lanczos or Arnoldi iteration
real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

* ``nev``: Number of eigenvalues
* ``ncv``: Number of Krylov vectors used in the computation; should satisfy ``nev+1 <= ncv <
  Note that these restrictions limit the input matrix ``A`` to be of dimension at least 2.
* ``which``: type of eigenvalues to compute. See the note below.

  ========= ================================================================================
  ``which`` type of eigenvalues
```

```
=========  ===============================================================================
``:LM``    eigenvalues of largest magnitude (default)
``:SM``    eigenvalues of smallest magnitude
``:LR``    eigenvalues of largest real part
``:SR``    eigenvalues of smallest real part
``:LI``    eigenvalues of largest imaginary part (nonsymmetric or complex ``A`` only)
``:SI``    eigenvalues of smallest imaginary part (nonsymmetric or complex ``A`` only)
``:BE``    compute half of the eigenvalues from each end of the spectrum, biased in favor o
=========  ===============================================================================

* ``tol``: tolerance (:math:`tol \le 0.0` defaults to ``DLAMCH('EPS')``)
* ``maxiter``: Maximum number of iterations (default = 300)
* ``sigma``: Specifies the level shift used in inverse iteration. If ``nothing`` (default),
* ``ritzvec``: Returns the Ritz vectors ``v`` (eigenvectors) if ``true``
* ``v0``: starting vector from which to start the iterations

``eigs`` returns the ``nev`` requested eigenvalues in ``d``, the corresponding Ritz vectors

.. note:: The ``sigma`` and ``which`` keywords interact: the description of eigenvalues sear

   =============== ================================== ==================================
   ``sigma``       iteration mode                     ``which`` refers to the problem
   =============== ================================== ==================================
   ``nothing``     ordinary (forward)                 :math:`Av = Bv\lambda`
   real or complex inverse with level shift ``sigma`` :math:`(A - \sigma B )^{-1}B = v\nu`
   =============== ================================== ==================================

..  eigs(A; nev=6, ncv=max(20,2*nev+1), which="LM", tol=0.0, maxiter=300, sigma=nothing, rit

Computes eigenvalues ``d`` of ``A`` using Lanczos or Arnoldi iterations for
real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

* ``nev``: Number of eigenvalues
* ``ncv``: Number of Krylov vectors used in the computation; should satisfy ``nev+1 <= ncv <
  Note that these restrictions limit the input matrix ``A`` to be of dimension at least 2.
* ``which``: type of eigenvalues to compute. See the note below.

  =========  ===============================================================================
  ``which``  type of eigenvalues
  =========  ===============================================================================
  ``:LM``    eigenvalues of largest magnitude (default)
  ``:SM``    eigenvalues of smallest magnitude
  ``:LR``    eigenvalues of largest real part
  ``:SR``    eigenvalues of smallest real part
  ``:LI``    eigenvalues of largest imaginary part (nonsymmetric or complex ``A`` only)
  ``:SI``    eigenvalues of smallest imaginary part (nonsymmetric or complex ``A`` only)
  ``:BE``    compute half of the eigenvalues from each end of the spectrum, biased in favor o
  =========  ===============================================================================

* ``tol``: tolerance (:math:`tol \le 0.0` defaults to ``DLAMCH('EPS')``)
```

```
* ``maxiter``: Maximum number of iterations (default = 300)
* ``sigma``: Specifies the level shift used in inverse iteration. If ``nothing`` (default),
* ``ritzvec``: Returns the Ritz vectors ``v`` (eigenvectors) if ``true``
* ``v0``: starting vector from which to start the iterations

``eigs`` returns the ``nev`` requested eigenvalues in ``d``, the corresponding Ritz vectors

.. note:: The ``sigma`` and ``which`` keywords interact: the description of eigenvalues sear

   =============== ================================== ==================================
   ``sigma``       iteration mode                     ``which`` refers to eigenvalues of
   =============== ================================== ==================================
   ``nothing``     ordinary (forward)                 :math:`A`
   real or complex inverse with level shift ``sigma`` :math:`(A - \sigma I )^{-1}`
   =============== ================================== ==================================
```

In [11]:
```
λeigs,Ueigs=eigs(A; nev=k, which=:LM, ritzvec=true, v0=x)
```

Out[11]:
```
([49.95446200130703,5.6118563334377844,-5.4800270872869215,-5.320761977199483,5.233
100x10 Array{Float64,2}:
 -0.0986278   0.134598    -0.0592035    ...   0.0108489   -0.00178483
 -0.0945524   0.26648     -0.000670684       -0.110458     0.09453
 -0.103228    0.0926254   -0.0480105          0.119675     0.0350439
 -0.105208   -0.208299     0.0158513          0.0410996    0.006113
 -0.099769   -0.0768202   -0.0790446          0.0515988   -0.0490355
 -0.102818   -0.113737     0.0162617    ...   0.008946     0.0126743
 -0.0977051  -0.00423835   0.212545           0.0658163    0.186496
 -0.105094   -0.181751     0.00995656         0.0854893    0.0868757
 -0.0933777  -0.0282255    0.0800883          0.0955286    0.103254
 -0.0970876   0.227432    -0.0268181          0.248907    -0.0290266
 -0.0974904   0.0228273    0.0169435    ...  -0.350786    -0.0139009
 -0.099804   -0.0176286   -0.0683804          0.00867715   0.012251
 -0.103317    0.0962935    0.149464           0.0485369    0.125337
   ⋮                                    ⋱
 -0.106478   -0.00313634   0.0389816          0.140828    -0.263224
 -0.104762    0.0378943    0.089783          -0.152489    -0.100868
 -0.105547   -0.0901734    0.022375     ...  -0.0189307    0.0657791
 -0.100194    0.0663129    0.125872           0.0479257   -0.0903578
 -0.107939    0.103618    -0.0489479         -0.180851     0.0299789
 -0.101306    0.190371     0.00048028        -0.0447766    0.0862519
 -0.0907318  -0.0951056   -0.0407354          0.0450873   -0.0705207
 -0.101851   -0.0449658   -0.105109     ...   0.133177     0.129686
 -0.100278   -0.05914     -0.0583337         -0.159586    -0.171006
 -0.108299    0.156876    -0.0105502          0.150995    -0.0767903
 -0.0936431   0.0366752   -0.0892431         -0.00559465  -0.101664
 -0.113033    0.0142365    0.174505           0.0724342    0.141767   ,

10,13,119,[0.165763,-0.0516391,0.500336,0.0918396,-0.220748,-0.261849,0.228609,-0.1
```

In [12]:
```
[sort(λ) sort(λeigs)]
```

Out[12]:
```
10x2 Array{Float64,2}:
 -5.35229   -5.48003
```

```
  -4.21933    -5.32076
  -3.19072    -5.04457
  -1.47379    -4.88439
   0.0234263   4.91068
   1.86636     4.95708
   3.16653     5.12328
   4.90863     5.23373
   5.46043     5.61186
  49.9545     49.9545
```

```
In [13]: sort(abs(λeig),rev=true)[1:k]
```

```
Out[13]: 10-element Array{Float64,1}:
  49.9545
   5.61186
   5.48003
   5.32076
   5.23373
   5.12328
   5.04457
   4.95708
   4.91068
   4.88439
```

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

```
In [14]: for i=1:k
             println(minabs(λeig-λ[i]))
         end
```

```
0.031526247053832
0.04038994453728417
0.011292078411404116
0.062240844138032436
0.01308176758937217
0.026173978684944643
0.005715112079596629
0.002050328510502908
0.1514301565936531
2.842170943040401e-14
```

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

```
In [15]: λall,Uall,Xall,μall=myLanczos(A,x,100)
```

```
Out[15]: ([-5.48003,-5.32076,-5.04457,-4.88439,-4.56518,-4.53665,-4.47582,-4.25972,-4.12503
         100x100 Array{Float64,2}:
           0.0592035    -0.0891002    ...   -0.0699932    -0.134598     0.0986278
           0.000670684   0.0262451          -0.082064     -0.26648      0.0945524
```

```
  0.0480105     0.0911351       0.0869701   -0.0926254   0.103228
 -0.0158513    -0.102907       -0.0109883    0.208299    0.105208
  0.0790446     0.0940853      -0.113809     0.0768202   0.099769
 -0.0162617    -0.123512    ...   0.0782959    0.113737    0.102818
 -0.212545     -0.0693098      -0.0765868    0.00423835  0.0977051
 -0.00995656   -0.0104302       0.00284639   0.181751    0.105094
 -0.0800883    -0.113806        0.0829726    0.0282255   0.0933777
  0.0268181    -0.00923486     -0.0215701   -0.227432    0.0970876
 -0.0169435     0.116483    ...  -0.107567    -0.0228273   0.0974904
  0.0683804    -0.148453       -0.0525417    0.0176286   0.099804
 -0.149464      0.122749       -0.0866675   -0.0962935   0.103317
  ⋮                         ⋱
 -0.0389816     0.106494       -0.0820755    0.00313634  0.106478
 -0.089783     -0.0512802      -0.128361    -0.0378943   0.104762
 -0.022375      0.0771865   ...   0.121995     0.0901734   0.105547
 -0.125872     -0.0588563       0.170807    -0.0663129   0.100194
  0.0489479     0.0645721      -0.14141     -0.103618    0.107939
 -0.00048028   -0.0392894      -0.0487408   -0.190371    0.101306
  0.0407354    -0.0203691       0.0304066    0.0951056   0.0907318
  0.105109     -0.0729856   ...   0.107971     0.0449658   0.101851
  0.0583337    -0.223372        0.00271593   0.05914     0.100278
  0.0105502    -0.073987       -0.039574    -0.156876    0.108299
  0.0892431    -0.106993        0.212378    -0.0366752   0.0936431
 -0.174505      0.0014195      -0.143662    -0.0142365   0.113033 ,

100x100 Array{Float64,2}:
 0.0554417    0.114285    -0.140432     ...  -0.0151959    0.129368
 0.0579906    0.105743    -0.225334           0.0695804   -0.0801958
 0.0611571    0.104891    -0.0133489          0.0220813    0.0255989
 0.086221     0.0596919    0.035153           0.0753488   -0.247913
 0.0795725    0.0665293   -0.0681834          0.0245431   -0.0499233
 0.136564    -0.0424879    0.106513     ...  -0.00980785  -0.0230636
 0.0735298    0.051477     0.261911           0.0988285    0.00222713
 0.172646    -0.0996042    0.0429803         -0.198679    -0.0228543
 0.154959    -0.0903062    0.0246825          0.0150503    0.0546877
 0.0471032    0.121375    -0.0744087          0.0663328   -0.0593263
 0.113582     0.00315446  -0.106804     ...  -0.0983546   -0.00791731
 0.000501112  0.212918    -0.0749958         -0.280243    -0.0515947
 0.136857    -0.0238376   -0.161593           0.0950437   -0.0442697
 ⋮                                      ⋱
 0.126114    -0.0138871    0.0688412         -0.0808086    0.0418425
 0.108062     0.0146213    0.0938811          0.0475039    0.0517171
 0.0445229    0.142419    -0.0525868    ...  -0.0152275   -0.0417183
 0.104384     0.0169799    0.0263259         -0.0183862    0.167778
 0.160958    -0.0692524    0.000101742        0.160276     0.00405783
 0.072694     0.0830557   -0.0602781         -0.0586542   -0.0205669
 0.0922272    0.0190697    0.0191022         -0.0455877    0.0639854
 0.117464     0.00357924  -0.0916713    ...  -0.00147318   0.0680306
 0.103506     0.017213     0.044306          -0.026247     0.0522335
 0.134201    -0.0218748    0.0338348         -0.121995    -0.0195552
 0.165831    -0.115479     0.121573          -0.037278    -0.0651084
```

```
 0.0972863     0.0580987    -0.0107879       -0.03544      -0.104446   ,

1.2466788127333224e-11)
```

```
In [16]: norm(A*Uall-Uall*diagm(λall)), norm(λeig-λall)

Out[16]: (1.250056955449801e-11,4.96737831990544e-14)
```

5.9.4 Operator version

One can use Lanczos method with operator which, given vector `x`, returns the product `A*x`. The principle is illustrated using the approach from the file `test/runtests.jl` from the package IterativeSolvers.jl.

Alternative is to use function `LinearMap()` from the package LinearMaps.jl

```
In [17]: type MyOp{T}
             buf::Matrix{T}
         end
         import Base: *, size, eltype, issym
         *(A::MyOp, x::AbstractVector) = A.buf*x
         size(A::MyOp, i) = size(A.buf, i)
         size(A::MyOp) = size(A.buf)
         eltype(A::MyOp) = eltype(A.buf)
         issym(A::MyOp) = issym(A.buf)

Out[17]: issym (generic function with 12 methods)
```

```
In [18]: B=MyOp(A)

Out[18]: MyOp{Float64}(100x100 Array{Float64,2}:
          0.646567    0.625402    0.502556     …  0.67409    0.32705     0.911176
          0.625402    0.785491    0.318734        0.964472   0.941207    0.720889
          0.502556    0.318734    0.874499        0.656513   0.992459    0.735653
          0.723478    0.063639    0.317501        0.416575   0.817869    0.714633
          0.245532    0.0823312   0.470855        0.998946   0.180076    0.335151
          0.104814    0.185622    0.80162      …  0.48265    0.0360261   0.959132
          0.284229    0.447635    0.185931        0.932729   0.659527    0.0859519
          0.621183    0.427134    0.241209        0.466468   0.660079    0.467693
          0.0165064   0.139989    0.472247        0.541534   0.119216    0.545585
          0.971307    0.302655    0.903899        0.77654    0.835625    0.733913
          0.49579     0.612132    0.00732402   …  0.846969   0.213473    0.382562
          0.214956    0.422308    0.385659        0.220166   0.772311    0.4311
          0.251386    0.697407    0.97089         0.994728   0.944495    0.0795707
          ⋮                                    ⋱
          0.400398    0.833964    0.94831         0.703164   0.988576    0.592791
          0.0606662   0.887372    0.977059        0.459312   0.392263    0.684222
          0.0355719   0.0968726   0.874157     …  0.94561    0.665153    0.411823
          0.0451803   0.479386    0.703112        0.610684   0.231819    0.792708
          0.824263    0.485306    0.12827         0.820904   0.362694    0.977492
          0.963972    0.0642148   0.56522         0.676011   0.0812902   0.769816
          0.40502     0.0319475   0.867078        0.383948   0.172502    0.722662
```

```
 0.798088   0.0450127  0.752206     ...  0.483946  0.951321   0.501643
 0.291267   0.863284   0.598341         0.209829  0.212784   0.739859
 0.67409    0.964472   0.656513         0.565751  0.092252   0.291205
 0.32705    0.941207   0.992459         0.092252  0.474853   0.380993
 0.911176   0.720889   0.735653         0.291205  0.380993   0.811761 )
```

In [19]:
```
λB,UB=eigs(B; nev=k, which=:LM, ritzvec=true, v0=x)
```

Out[19]:
```
([49.95446200130703,5.6118563334377844,-5.4800270872869215,-5.320761977199483,5.233
100x10 Array{Float64,2}:
 -0.0986278   0.134598    -0.0592035     ...   0.0108489    -0.00178483
 -0.0945524   0.26648     -0.000670684      -0.110458      0.09453
 -0.103228    0.0926254   -0.0480105         0.119675      0.0350439
 -0.105208   -0.208299     0.0158513         0.0410996     0.006113
 -0.099769   -0.0768202   -0.0790446         0.0515988    -0.0490355
 -0.102818   -0.113737     0.0162617     ...   0.008946      0.0126743
 -0.0977051  -0.00423835   0.212545          0.0658163     0.186496
 -0.105094   -0.181751     0.00995656        0.0854893     0.0868757
 -0.0933777  -0.0282255    0.0800883         0.0955286     0.103254
 -0.0970876   0.227432    -0.0268181         0.248907     -0.0290266
 -0.0974904   0.0228273    0.0169435     ...  -0.350786     -0.0139009
 -0.099804   -0.0176286   -0.0683804         0.00867715    0.012251
 -0.103317    0.0962935    0.149464          0.0485369     0.125337
  ⋮                                       ⋱
 -0.106478   -0.00313634   0.0389816         0.140828     -0.263224
 -0.104762    0.0378943    0.089783         -0.152489     -0.100868
 -0.105547   -0.0901734    0.022375      ...  -0.0189307     0.0657791
 -0.100194    0.0663129    0.125872          0.0479257    -0.0903578
 -0.107939    0.103618    -0.0489479        -0.180851      0.0299789
 -0.101306    0.190371     0.00048028       -0.0447766     0.0862519
 -0.0907318  -0.0951056   -0.0407354         0.0450873    -0.0705207
 -0.101851   -0.0449658   -0.105109      ...   0.133177      0.129686
 -0.100278   -0.05914     -0.0583337        -0.159586     -0.171006
 -0.108299    0.156876    -0.0105502         0.150995     -0.0767903
 -0.0936431   0.0366752   -0.0892431        -0.00559465   -0.101664
 -0.113033    0.0142365    0.174505          0.0724342     0.141767   ,

10,13,119,[0.165763,-0.0516391,0.500336,0.0918396,-0.220748,-0.261849,0.228609,-0.1
```

In [20]:
```
λeigs-λB
```

Out[20]:
```
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
```

```
In [22]: # With LinearMaps
         using LinearMaps

In [23]: methods(LinearMap)

Out[23]: # 5 methods for generic function "LinearMap":
         LinearMap{T}(A::Union{AbstractArray{T,2},LinearMaps.AbstractLinearMap{T}}) at /home
         LinearMap(f::Function, M::Int64) at /home/slap/.julia/v0.4/LinearMaps/src/LinearMap
         LinearMap(f::Function, M::Int64, N::Int64) at /home/slap/.julia/v0.4/LinearMaps/src
         LinearMap(f::Function, eltype::Type{T}, M::Int64) at /home/slap/.julia/v0.4/LinearM
         LinearMap(f::Function, eltype::Type{T}, M::Int64, N::Int64) at /home/slap/.julia/v0

In [24]: # Operator from the matrix
         C=LinearMap(A)

Out[24]: LinearMaps.WrappedMap{Float64}(100x100 Array{Float64,2}:
          0.646567   0.625402   0.502556    ...  0.67409    0.32705    0.911176
          0.625402   0.785491   0.318734       0.964472  0.941207   0.720889
          0.502556   0.318734   0.874499       0.656513  0.992459   0.735653
          0.723478   0.063639   0.317501       0.416575  0.817869   0.714633
          0.245532   0.0823312  0.470855       0.998946  0.180076   0.335151
          0.104814   0.185622   0.80162     ...  0.48265    0.0360261  0.959132
          0.284229   0.447635   0.185931       0.932729  0.659527   0.0859519
          0.621183   0.427134   0.241209       0.466468  0.660079   0.467693
          0.0165064  0.139989   0.472247       0.541534  0.119216   0.545585
          0.971307   0.302655   0.903899       0.77654   0.835625   0.733913
          0.49579    0.612132   0.00732402  ...  0.846969  0.213473   0.382562
          0.214956   0.422308   0.385659       0.220166  0.772311   0.4311
          0.251386   0.697407   0.97089        0.994728  0.944495   0.0795707
          ⋮                                 ⋱
          0.400398   0.833964   0.94831        0.703164  0.988576   0.592791
          0.0606662  0.887372   0.977059       0.459312  0.392263   0.684222
          0.0355719  0.0968726  0.874157    ...  0.94561    0.665153   0.411823
          0.0451803  0.479386   0.703112       0.610684  0.231819   0.792708
          0.824263   0.485306   0.12827        0.820904  0.362694   0.977492
          0.963972   0.0642148  0.56522        0.676011  0.0812902  0.769816
          0.40502    0.0319475  0.867078       0.383948  0.172502   0.722662
          0.798088   0.0450127  0.752206    ...  0.483946  0.951321   0.501643
          0.291267   0.863284   0.598341       0.209829  0.212784   0.739859
          0.67409    0.964472   0.656513       0.565751  0.092252   0.291205
          0.32705    0.941207   0.992459       0.092252  0.474853   0.380993
          0.911176   0.720889   0.735653       0.291205  0.380993   0.811761 ,true,true,true

In [25]: λC,UC=eigs(C; nev=k, which=:LM, ritzvec=true, v0=x)
         λeigs-λC

Out[25]: 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 [26]: f(x)=A*x

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

In [27]: D=LinearMap(f,n,issym=true)

Out[27]: FunctionMap{Float64}(f,100,100;ismutating=false,issym=true,ishermitian=true,isposde

In [28]: λD,UD=eigs(D, nev=k, which=:LM, ritzvec=true, v0=x)
         λeigs-λD

Out[28]: 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
```

5.9.5 Sparse matrices

```
In [29]: C=sprand(n,n,0.05) |> t -> t+t'

Out[29]: 100x100 sparse matrix with 958 Float64 entries:
                 [18 ,   1]  =  0.421797
                 [33 ,   1]  =  0.683238
                 [36 ,   1]  =  0.497261
                 [50 ,   1]  =  0.768738
                 [61 ,   1]  =  0.126499
                 [91 ,   1]  =  0.0383286
                 [100,   1]  =  0.382148
                 [14 ,   2]  =  0.156032
                 [15 ,   2]  =  1.46461
                 [18 ,   2]  =  0.749037
                 ⋮
                 [39 ,  99]  =  1.16761
                 [59 ,  99]  =  0.0518335
                 [63 ,  99]  =  0.941817
```

```
       [90 ,  99]  =  0.247089
       [93 ,  99]  =  0.739849
       [1  , 100]  =  0.382148
       [2  , 100]  =  0.420669
       [22 , 100]  =  0.830799
       [30 , 100]  =  0.00513943
       [46 , 100]  =  0.605078
       [56 , 100]  =  0.29828
```

In [30]: `issym(C)`

Out[30]: `true`

In [31]: `eigs(C; nev=k, which=:LM, ritzvec=true, v0=x)`

Out[31]:
```
([5.75035290520615,-3.75272830445963,3.599167050390188,-3.5672816776275615,-3.43825
100x10 Array{Float64,2}:
 -0.0568613   0.0313642   0.0218171   ...   0.0715168   -0.105739
 -0.148207    0.0177587   0.218053          0.179457     0.0181708
 -0.0720305  -0.0631807   0.0223883         0.0032417   -0.004229
 -0.137368   -0.206325    0.15269           0.0223327   -0.00407251
 -0.190019    0.305452   -0.230547          0.0323472    0.125015
 -0.165947   -0.24769     0.0099424   ...   0.263271     0.172879
 -0.0824389  -0.0613767  -0.0435096         0.0467548    0.155994
 -0.121827   -0.0169714  -0.146714         -0.00550256   0.067811
 -0.125282    0.0166506  -0.261541         -0.178184    -0.178426
 -0.0974252   0.143441    0.0387801         0.0566182    0.117898
 -0.0751094   0.0698195  -0.0462562   ...   0.0404678    0.0408233
 -0.143482    0.016242   -0.0534555         0.211353    -0.179002
 -0.111092    0.19888     0.0902481         0.05193      0.0897668
  ⋮                                     ⋱
 -0.040382    0.0169914   0.0765816        -0.00874344   0.0618363
 -0.0496034   0.0422594  -0.013896         -0.0148572    0.0645763
 -0.0732564  -0.120627   -0.0307551   ...  -0.0513495    0.0396249
 -0.118364   -0.0433191  -0.139616         -0.120876     0.0675308
 -0.0640958  -0.0595567  -0.0247941         0.0908123   -0.0548542
 -0.0932615  -0.0057891  -0.064663          0.059493     0.156088
 -0.0958927   0.0875136  -0.0823829         0.0631593   -0.0376929
 -0.132818   -0.0204587   0.323165    ...  -0.0205479    0.0223358
 -0.0788443  -0.0905153  -0.0451236        -0.023231     0.00457104
 -0.0687472  -0.0333914  -0.112879          0.0590753   -0.0190283
 -0.0410418   0.0142648   0.0536162        -0.0494264   -0.0706246
 -0.0322752  -0.0114379   0.00873152       -0.0603988   -0.0622746 ,

10,14,122,[-0.283055,0.312621,-0.160701,-0.298127,-0.0611511,0.0450921,0.00884099,(
```

In [ ]:

6 Singular Value Decomposition - Definitions and Facts

6.1 Prerequisites

The reader should be familiar with basic linear algebra concepts and notebooks related to eigenvalue decomposition.

6.2 Competences

The reader should be able to undestand and check the facts about singular value decomposition.

6.3 Selected references

There are many excellent books on the subject. Here we list a few:
J. W. Demmel, Applied Numerical Linear Algebra
G. H. Golub and C. F. Van Loan, Matrix Computations
N. Higham, Accuracy and Stability of Numerical Algorithms
L. Hogben, ed., Handbook of Linear Algebra
G. W. Stewart, Matrix Algorithms, Vol. II: Eigensystems
L. N. Trefethen and D. Bau, III, Numerical Linear Algebra

6.4 Singular value problems

For more details and the proofs of the Facts below, see R. C. Li, Matrix Perturbation Theory and R. Mathias, Singular Values and Singular Value Inequalities and the references therein.

6.4.1 Definitions

Let $A \in \mathbb{C}^{m\times n}$ and let $q=\min\{m,n\}$.
The **singular value decomposition** (SVD) of A is $A=U\Sigma V^*$, where $U\in\mathbb{C}^{m\times m}$ and $V\in\mathbb{C}^{n\times n}$ are unitary, and $\Sigma=\operatorname{diag}(\sigma_1,\sigma_2,\ldots)\in\mathbb{R}^{m\times n}$ with all $\sigma_j\geq 0$.
Here σ_j is the **singular value**, $u_j\equiv U_{:,j}$ is the corresponding **left singular vector**, and $v_j\equiv V_{:,j}$ is the corresponding **right singular vector**.
The **set of singular values** is $sv(A)=\{\sigma_1,\sigma_2,\ldots,\sigma_q\}$.
We assume that singular values are ordered, $\sigma_1\geq\sigma_2\geq\cdots\sigma_q\geq 0$.
The **Jordan-Wielandt** matrix is the Hermitian matrix

$$J=\begin{bmatrix} 0 & A \\ A^* & 0\end{bmatrix}\in\mathbb{C}^{(m+n)\times(m+n)}.$$

6.4.2 Facts

There are many facts related to the singular value problem for general matrices. We state some basic ones:

1. If $A\in\mathbb{R}^{m\times n}$, then U and V are real.

2. Singular values are unique.

3. $\sigma_j(A^T) = \sigma_j(A^*) = \sigma_j(\bar{A}) = \sigma_j(A)$ for $j = 1, 2, \ldots, q$.

4. $Av_j = \sigma_j u_j$ and $A^* u_j = \sigma_j v_j$ for $j = 1, 2, \ldots, q$.

5. $A = \sigma_1 u_1 v_1^* + \sigma_2 u_2 v_2^* + \cdots + \sigma_q u_q v_q^*$.

6. *(Unitary invariance)* For any unitary $U \in \mathbb{C}^{m \times m}$ and $V \in \mathbb{C}^{n \times n}$, $sv(A) = sv(UAV)$.

7. There exist unitary matrices $U \in \mathbb{C}^{m \times m}$ and $V \in \mathbb{C}^{n \times n}$ such that $A = UBV$ if and only if $sv(A) = sv(B)$.

8. SVD of A is related to eigenvalue decompositions of Hermitian matrices $A^*A = V\Sigma^T\Sigma V^*$ and $AA^* = U\Sigma\Sigma^T U^*$. Thus, $\sigma_j^2(A) = \lambda_j(A^*A) = \lambda_j(AA^*)$ for $j = 1, 2, \ldots, q$.

9. The eigenvalues of Jordan-Wielandt matrix are $\pm\sigma_1(A), \pm\sigma_2(A), \cdots, \pm\sigma_q(A)$ together with $|m - n|$ zeros. The eigenvectors are obtained from an SVD of A. This relationship is used to deduce singular value results from the results for eigenvalues of Hermitian matrices.

10. $\text{trace}(|A|_{spr}) = \sum_{i=1}^{q} \sigma_i$, where $|A|_{spr} = (A^*A)^{1/2}$.

11. If A is square, then $|\det(A)| = \prod_{i=1}^{n} \sigma_i$.

12. If A is square, then A is singular $\Leftrightarrow \sigma_j(A) = 0$ for some j.

13. *(Min-max Theorem)* It holds:

$$\begin{aligned}\sigma_k &= \max_{\dim(W)=k} \min_{x \in W, \|x\|_2=1} \|Ax\|_2 \\ &= \min_{\dim(W)=n-k+1} \max_{x \in W, \|x\|_2=1} \|Ax\|_2.\end{aligned}$$

14. $\|A\|_2 = \sigma_1(A)$.

15. For $B \in \mathbb{C}^{m \times n}$,

$$|\text{trace}(AB^*)| \leq \sum_{j=1}^{q} \sigma_j(A)\sigma_j(B).$$

16. *(Interlace Theorems)* Let B denote A with the one of its rows *or* columns deleted. Then

$$\sigma_{j+1}(A) \leq \sigma_j(B) \leq \sigma_j(A), \quad j = 1, \ldots, q-1.$$

Let B denote A with the one of its rows *and* columns deleted. Then

$$\sigma_{j+2}(A) \leq \sigma_j(B) \leq \sigma_j(A), \quad j = 1, \ldots, q-2.$$

17. *(Weyl Inequalities)* For $B \in \mathbb{C}^{m \times n}$, it holds:

$$\begin{aligned}&\sigma_{j+k-1}(A+B) \leq \sigma_j(A) + \sigma_k(B), \quad j + k \leq n+1, \\ &\sum_{j=1}^{k} \sigma_j(A+B) \leq \sum_{j=1}^{k} \sigma_j(A) + \sum_{j=1}^{k} \sigma_j(A), \quad k = 1, \ldots, q.\end{aligned}$$

6.4.3 Example - Symbolic computation

```
In [1]: using SymPy

In [2]: A=[  3   2   1
            -5  -1  -4
             5   0   2]

Out[2]: 3x3 Array{Int64,2}:
          3   2   1
         -5  -1  -4
          5   0   2

In [3]: @vars x

Out[3]: (x,)

In [4]: B=A'*A

Out[4]: 3x3 Array{Int64,2}:
         59  11  33
         11   5   6
         33   6  21

In [5]: # Characteristic polynomial p_B(λ)
        p(x)=simplify(det(B-x*I))
        p(x)

Out[5]:
```

$$-x^3 + 85x^2 - 393x + 441$$

```
In [6]: λ=map(Rational,solve(p(x),x))

Out[6]: 3-element Array{Rational{Int64},1}:
                        3//1
         2064549086305011//1125899906842624
         5641202704674385//70368744177664

In [7]: V=Array(Any,3,3)
        for j=1:3
            V[:,j]=nullspace(B-λ[j]*I)
        end
        V

Out[7]: 3x3 Array{Any,2}:
         -3.2754e-7  -0.519818  -0.854277
          0.948684    0.270146  -0.164381
         -0.316227    0.810438  -0.493142

In [8]: U=Array(Any,3,3)
        for j=1:3
            U[:,j]=nullspace(A*A'-λ[j]*I)
        end
        U
```

```
Out[8]: 3x3 Array{Any,2}:
          0.912871  -0.154138  -0.378032
          0.182574  -0.67409    0.71573
         -0.365148  -0.722388  -0.587215
```

```
In [9]: σ=sqrt(λ)
```

```
Out[9]: 3-element Array{Float64,1}:
         1.73205
         1.35414
         8.95356
```

```
In [10]: A-U*diagm(σ)*V'
```

```
Out[10]: 3x3 Array{Any,2}:
           2.02284e-7   3.05213e-7   9.51424e-7
           9.53544e-7  -6.66283e-7  -7.23387e-7
          -7.64795e-7   1.85427e-8   3.64772e-7
```

```
In [11]: svd(A)
```

```
Out[11]: (
         3x3 Array{Float64,2}:
          -0.378032  -0.912871  -0.154137
           0.71573   -0.182574  -0.67409
          -0.587215   0.365148  -0.722388,

         [8.95356420958337,1.7320508075688772,1.3541373434285466],
         3x3 Array{Float64,2}:
          -0.854277   0.0        -0.519818
          -0.164381  -0.948683    0.270146
          -0.493143   0.316228    0.810438)
```

6.4.4 Example - Random complex matrix

```
In [12]: m=5
         n=3
         q=min(m,n)
         A=rand(m,n)+im*rand(m,n)
```

```
Out[12]: 5x3 Array{Complex{Float64},2}:
          0.840481+0.754007im    0.789838+0.26694im     0.15832+0.353723im
          0.978061+0.764122im    0.037961+0.912471im  0.550265+0.914916im
          0.175202+0.685971im  0.00701676+0.263729im  0.901928+0.294551im
          0.298071+0.627254im    0.205681+0.161038im  0.394636+0.682877im
          0.560192+0.202763im    0.246736+0.729301im  0.321625+0.702612im
```

```
In [13]: U,σ,V=svd(A, thin=false)
```

```
Out[13]: (
         5x5 Array{Complex{Float64},2}:
            -0.3691-0.239435im   0.533531+0.17468im    ...  -0.0300464+0.128464im
```

```
 -0.417586-0.482031im  0.0732014-0.0735096im       -0.47182-0.367117im
  -0.24535-0.234908im  -0.677775+0.272126im        0.141541+0.273095im
 -0.205844-0.286222im   -0.15569-0.163952im       0.0198208+0.0437499im
 -0.289897-0.275663im  0.0900631-0.28532im         0.718303+0.111017im ,

[2.8788636227103237,0.9414389301313411,0.8055571946914614],
3x3 Array{Complex{Float64},2}:
 -0.670689-0.0im          0.538362-0.0im          0.510238-0.0im
  -0.42927+0.157316im     0.240547-0.0404805im  -0.818064+0.249498im
 -0.579403+0.0738135im  -0.803844+0.0670857im  0.0865478+0.0262416im)
```

```
In [14]: norm(A-U[:,1:q]*diagm(σ)*V'), norm(U'*U-I), norm(V'*V-I)
```

```
Out[14]: (1.4687615521463284e-15,5.37008467320231e-16,4.183902309112554e-16)
```

```
In [15]: # Fact 4
         @show k=rand(1:q)
         norm(A*V[:,k]-σ[k]*U[:,k],Inf), norm(A'*U[:,k]-σ[k]*V[:,k],Inf)
```

```
k = rand(1:q) = 2
```

```
Out[15]: (5.578801654593729e-16,7.816041058999314e-16)
```

```
In [16]: λV,V1=eig(A'*A)
```

```
Out[16]: ([0.6489223939191779,0.8863072591668464,8.287855758164794],
         3x3 Array{Complex{Float64},2}:
           -0.488287+0.148051im  -0.536497-0.044774im   0.665311+0.084758im
            0.710475-0.476133im   -0.24308+0.0203347im  0.445709-0.101806im
          -0.0904386-0.0im        0.806638+0.0im        0.584086+0.0im      )
```

```
In [17]: sqrt(λV)
```

```
Out[17]: 3-element Array{Float64,1}:
          0.805557
          0.941439
          2.87886
```

```
In [18]: λU,U1=eig(A*A')
```

```
Out[18]: ([-6.106226640262088e-16,7.097554593628281e-16,0.6489223939191757,0.88630725916684
         5x5 Array{Complex{Float64},2}:
           0.108369+0.167766im   -0.245751-0.162165im   ...   -0.43247+0.0808321im
          -0.472394-0.292261im     -0.2476-0.0790917im      -0.634777-0.0615589im
           0.295752+0.113347im   -0.340116+0.265275im       -0.339672-0.00116311im
          -0.273781+0.0549008im   0.734847+0.238574im       -0.346404-0.0655717im
           0.687932+0.0im          0.25056+0.0im            -0.400038-0.0im        )
```

```
In [19]: V,V1
```

```
Out[19]: (
         3x3 Array{Complex{Float64},2}:
          -0.670689-0.0im         0.538362-0.0im          0.510238-0.0im
           -0.42927+0.157316im    0.240547-0.0404805im  -0.818064+0.249498im
          -0.579403+0.0738135im  -0.803844+0.0670857im  0.0865478+0.0262416im,

         3x3 Array{Complex{Float64},2}:
           -0.488287+0.148051im  -0.536497-0.044774im   0.665311+0.084758im
            0.710475-0.476133im   -0.24308+0.0203347im  0.445709-0.101806im
          -0.0904386-0.0im        0.806638+0.0im        0.584086+0.0im      )
```

Explain the non-uniqueness of U and V!

```
In [20]: # Jordan-Wielandt matrix
         J=[zeros(A*A') A; A' zeros(A'*A)]
```

```
Out[20]: 8x8 Array{Complex{Float64},2}:
              0.0+0.0im           0.0+0.0im       ...   0.15832+0.353723im
              0.0+0.0im           0.0+0.0im            0.550265+0.914916im
              0.0+0.0im           0.0+0.0im            0.901928+0.294551im
              0.0+0.0im           0.0+0.0im            0.394636+0.682877im
              0.0+0.0im           0.0+0.0im            0.321625+0.702612im
          0.840481-0.754007im  0.978061-0.764122im  ...       0.0+0.0im
          0.789838-0.26694im   0.037961-0.912471im            0.0+0.0im
           0.15832-0.353723im  0.550265-0.914916im            0.0+0.0im
```

```
In [21]: λJ,UJ=eig(J)
```

```
Out[21]: ([-2.878863622710325,-0.9414389301313433,-0.8055571946914597,-5.3104555203255555e-1
         8x8 Array{Complex{Float64},2}:
           0.237505+0.200932im    0.365684+0.154465im    ...  -0.237505-0.200932im
           0.249836+0.37543im    0.0559048-0.0474942im       -0.249836-0.37543im
           0.151106+0.186698im   -0.493602+0.151897im        -0.151106-0.186698im
            0.11881+0.219161im   -0.100066-0.124686im         -0.11881-0.219161im
           0.178711+0.219266im   0.0802427-0.195756im        -0.178711-0.219266im
          -0.470446-0.0599329im  -0.379361-0.03166im     ...  -0.470446-0.0599329im
          -0.315164+0.0719879im  -0.171884+0.0143788im       -0.315164+0.0719879im
          -0.413011-0.0im         0.570379+0.0im             -0.413011-0.0im       )
```

6.4.5 Example - Random real matrix

```
In [22]: m=8
         n=5
         q=min(m,n)
         A=rand(-9:9,m,n)
```

```
Out[22]: 8x5 Array{Int64,2}:
           8  -3   1   9  -7
          -9   8  -6   1  -1
          -5   8   1  -3  -9
           4   8  -6   9   7
           9   4  -6   8   2
```

```
 -4   7   3   6  -7
 -4   7  -8  -4  -2
 -9   5  -1   0   8
```

In [23]:
```
U,σ,V=svd(A)
```

Out[23]:
```
(
8x5 Array{Float64,2}:
 -0.426362    0.217265   -0.471716    -0.0774665   0.474191
  0.510042    0.216984   -0.0733038    0.0183391   0.617511
  0.375413   -0.0664926  -0.525767     0.13404    -0.521785
  0.0143749   0.699875    0.208505    -0.125711   -0.276064
 -0.222732    0.595224   -0.00288216   0.247566   -0.148558
  0.201234    0.166994   -0.547027    -0.504665   -0.0733812
  0.405197    0.143128   -0.0413489    0.641799    0.126587
  0.403288    0.0938272   0.388907    -0.481676    0.0432935,

[24.043444540388922,21.601117837961112,17.8635139679196,10.289332947275364,4.83000
5x5 Array{Float64,2}:
 -0.743685    0.286527  -0.0461132   0.394274    -0.455254
  0.575968    0.517197  -0.218042   -0.00120866  -0.594327
 -0.20389    -0.44718   -0.195403   -0.675583    -0.513676
 -0.271148    0.641724  -0.224185   -0.566318     0.37907
 -0.0100566   0.196544   0.92838    -0.25965     -0.178778)
```

In [24]:
```
# Fact 10
trace(sqrtm(A'*A)), sum(σ)
```

Out[24]:
```
(78.62741747177996,78.62741747177978)
```

In [25]:
```
# Fact 11
B=rand(n,n)
det(B), prod(svdvals(B))
```

Out[25]:
```
(0.035587238998927 67,0.03558723899892767)
```

In [26]:
```
# Fact 14
norm(A), σ[1]
```

Out[26]:
```
(24.04344454038893,24.043444540388922)
```

In [27]:
```
# Fact 15
B=rand(m,n)
abs(trace(A*B')), sum(svdvals(A)⋅svdvals(B))
```

Out[27]:
```
(11.518737671982173,134.5922565166571)
```

In [28]:
```
# Interlace Theorems (repeat several times)
j=rand(1:q)
σBrow=svdvals(A[[1:j-1;j+1:m],:])
σBcol=svdvals(A[:,[1:j-1;j+1:n]])
j, σ, σBrow, σBcol
```

```
Out[28]: (2,[24.043444540388922,21.601117837961112,17.8635139679196,10.289332947275364,4.830
```

```
In [29]: σ[1:end].>=σBrow, σ[1:end-1].>=σBcol, σ[2:end].<=σBrow[1:end-1], σ[2:end].<=σBco
```

```
Out[29]: (Bool[true,true,true,true,true],Bool[true,true,true,true],Bool[true,true,true,true]
```

```
In [30]: # Weyl Inequalities
         B=rand(m,n)
         μ=svdvals(B)
         γ=svdvals(A+B)
         [γ σ μ]
```

```
Out[30]: 5x3 Array{Float64,2}:
          23.9055    24.0434    3.0671
          22.6926    21.6011    1.11292
          17.7938    17.8635    0.951415
          10.269     10.2893    0.645251
           4.65344    4.83001   0.558815
```

```
In [31]: @show k=rand(1:q)
         sum(γ[1:k]),sum(σ[1:k])+sum(μ[1:k])

k = rand(1:q) = 3
```

```
Out[31]: (64.39198222605349,68.639507885397)
```

6.5 Matrix approximation

Let $A = U\Sigma V^*$, let $\tilde{\Sigma}$ be equal to Σ except that $\tilde{\Sigma}_{jj} = 0$ for $j > k$, and let $\tilde{A} = U\tilde{\Sigma}V^*$. Then $\text{rank}(\tilde{A}) \leq k$ and

$$\min\{\|A - B\|_2 : \text{rank}(B) \leq k\} = \|A - \tilde{A}\|_2 = \sigma_{k+1}(A)$$

$$\min\{\|A - B\|_F : \text{rank}(B) \leq k\} = \|A - \tilde{A}\|_F = \left(\sum_{j=k+1}^{q} \sigma_j^2(A)\right)^{1/2}.$$

This is the **Eckart-Young-Mirsky Theorem**.

```
In [32]: @show k=rand(1:q-1)
         B=U*diagm([σ[1:k];zeros(q-k)])*V'

k = rand(1:q - 1) = 3
```

```
Out[32]: 8x5 Array{Float64,2}:
           9.35696  -1.63975   1.638      7.6804     -6.7975
          -7.71657   9.77286  -4.34044   -0.0237447  -0.417785
          -6.69112   6.50383   0.637169  -1.2636     -9.09246
           3.90295   7.20597  -7.55878    8.77293     6.42577
           7.66901   3.57663  -4.64767    9.71457     2.53312
          -2.11402   6.78308  -0.690142   3.19366    -8.41164
          -6.32531   7.37136  -3.22459   -0.491988   -0.176049
          -6.95073   5.11829  -4.24086   -2.88601     6.75053
```

```
In [33]: norm(A-B), σ[k+1]

Out[33]: (10.289332947275367,10.289332947275364)

In [34]: vecnorm(A-B),vecnorm(σ[k+1:q])

Out[34]: (11.366589264229676,11.36658926422967)

In [ ]:
```

7 Singular Value Decomposition - Perturbation Theory

7.1 Prerequisites

The reader should be familiar with eigenvalue decomposition, singular value decompostion, and perturbation theory for eigenvalue decomposition.

7.2 Competences

The reader should be able to understand and check the facts about perturbations of singular values and vectors.

7.3 Peturbation bounds

For more details and the proofs of the Facts below, see R.-C. Li, Matrix Perturbation Theory, and the references therein.

7.3.1 Definitions

Let $A \in \mathbb{C}^{m\times n}$ and let $A = U\Sigma V^*$ be its SVD.

The set of A's singular values is $sv(B) = \{\sigma_1, \sigma_2, \ldots)$, with $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$, and let $sv_{ext}(B) = sv(B)$ unless $m > n$ for which $sv_{ext}(B) = sv(B) \cup \{0, \ldots, 0\}$ (additional $|m-n|$ zeros).

Triplet $(u, \sigma, v) \in \times\mathbb{C}^m \times \mathbb{R} \times \mathbb{C}^n$ is a **singular triplet** of A if $\|u\|_2 = 1$, $\|v\|_2 = 1$, $\sigma \geq 0$, and $Av = \sigma u$ and $A^*u = \sigma v$.

$\tilde{A} = A + \Delta A$ is a **perturbed matrix**, where ΔA is **perturbation**. *The same notation is adopted to $\tilde{A}$, except all symbols are with tildes.*

Spectral condition number of A is $\kappa_2(A) = \sigma_{\max}(A)/\sigma_{\min}(A)$.

Let $X, Y \in \mathbb{C}^{n\times k}$ with $\operatorname{rank}(X) = \operatorname{rank}(Y) = k$. The **canonical angles** between their column spaces are $\theta_i = \cos^{-1}\sigma_i$, where σ_i are the singular values of $(Y^*Y)^{-1/2}Y^*X(X^*X)^{-1/2}$. The **canonical angle matrix** between X and Y is

$$\Theta(X, Y) = \operatorname{diag}(\theta_1, \theta_2, \ldots, \theta_k).$$

7.3.2 Facts

1. *(Mirsky)* $\|\Sigma - \tilde{\Sigma}\|_2 \leq \|\Delta A\|_2$ and $\|\Sigma - \tilde{\Sigma}\|_F \leq \|\Delta A\|_F$.

2. *(Residual bounds)* Let $\|\tilde{u}\|_2 = \|\tilde{v}\|_2 = 1$ and $\tilde{\mu} = \tilde{u}^*A\tilde{v}$. Let residuals $r = A\tilde{v} - \tilde{\mu}\tilde{u}$ and $s = A^*\tilde{u} - \tilde{\mu}\tilde{v}$, and let $\varepsilon = \max\{\|r\|_2, \|s\|_2\}$. Then $|\tilde{\mu} - \mu| \leq \varepsilon$ for some singular value μ of A.

3. The smallest error matrix δA for which $(\tilde{u}, \tilde{\mu}, \tilde{v})$ is a singular triplet of $\tilde{A}$ satisfies $\|\Delta A\|_2 = \varepsilon$.

4. Let μ be the closest singular value in $sv_{ext}(A)$ to $\tilde{\mu}$ and (u, μ, v) be the associated singular triplet, and let
$$\eta = \operatorname{gap}(\tilde{\mu}) = \min_{\mu\neq\sigma\in sv_{ext}(A)} |\tilde{\mu} - \sigma|.$$

If $\eta > 0$, then

$$|\tilde{\mu} - \mu| \leq \frac{\varepsilon^2}{\eta},$$
$$\sqrt{\sin^2\theta(u,\tilde{u}) + \sin^2\theta(v,\tilde{v})} \leq \frac{\sqrt{\|r\|_2^2 + \|s\|_2^2}}{\eta}.$$

5. Let
$$A = \begin{bmatrix} M & E \\ F & H \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} M & 0 \\ 0 & H \end{bmatrix},$$
where $M \in \mathbb{C}^{k\times k}$, and set $\eta = \min|\mu - \nu|$ over all $\mu \in sv(M)$ and $\nu \in sv_{ext}(H)$, and $\varepsilon = \max\{\|E\|_2, \|F\|_2\}$. Then
$$\max|\sigma_j - \tilde{\sigma}_j| \leq \frac{2\varepsilon^2}{\eta + \sqrt{\eta^2 + 4\varepsilon^2}}.$$

6. Let $m \geq n$ and let
$$\begin{bmatrix} U_1^* \\ U_2^* \end{bmatrix} A \begin{bmatrix} V_1 & V_2 \end{bmatrix} = \begin{bmatrix} A_1 & \\ & A_2 \end{bmatrix}, \quad \begin{bmatrix} \tilde{U}_1^* \\ \tilde{U}_2^* \end{bmatrix} \tilde{A} \begin{bmatrix} \tilde{V}_1 & \tilde{V}_2 \end{bmatrix} = \begin{bmatrix} \tilde{A}_1 & \\ & \tilde{A}_2 \end{bmatrix},$$
where $\begin{bmatrix} U_1 & U_2 \end{bmatrix}$, $\begin{bmatrix} V_1 & V_2 \end{bmatrix}$, $\begin{bmatrix} \tilde{U}_1 & \tilde{U}_2 \end{bmatrix}$, and $\begin{bmatrix} \tilde{V}_1 & \tilde{V}_2 \end{bmatrix}$ are unitary, and $U_1, \tilde{U}_1 \in \mathbb{C}^{m\times k}$, $V_1, \tilde{V}_1 \in \mathbb{C}^{n\times k}$. Set
$$R = A\tilde{V}_1 - \tilde{U}_1\tilde{A}_1, \quad S = A^*\tilde{U}_1 - \tilde{V}_1\tilde{A}_1.$$
Let $\eta = \min|\tilde{\mu} - \nu|$ over all $\tilde{\mu} \in sv(\tilde{A}_1)$ and $\nu \in sv_{ext}(A_2)$. If $\eta > 0$, then
$$\sqrt{\|\sin\Theta(U_1,\tilde{U}_1)\|_F^2 + \|\sin\Theta(V_1,\tilde{V}_1)\|_F^2} \leq \frac{\sqrt{\|R\|_F^2 + \|S\|_F^2}}{\eta}.$$

7.3.3 Example

In [1]:
```
m=8
n=5
k=min(m,n)
A=rand(-9:9,m,n)
```

Out[1]:
```
8x5 Array{Int64,2}:
 -5   7  -6   1  -6
 -7   7  -4   5   2
 -2   7  -9  -8   6
  1  -9   8  -9   5
 -4  -1   6  -2  -2
  3  -1   7   3  -1
  3   3  -1  -7  -5
  4  -3   1   3   0
```

In [2]:
```
ΔA=rand(m,n)/100
B=A+ΔA
```

```
Out[2]: 8x5 Array{Float64,2}:
 -4.99651   7.00853   -5.99439    1.0043    -5.99049
 -6.9933    7.00928   -3.99684    5.00413    2.00102
 -1.99078   7.00601   -8.99964   -7.99108    6.00944
  1.00361  -8.9954     8.00587   -8.99752    5.00248
 -3.99393  -0.992812   6.00234   -1.99216   -1.99733
  3.0097   -0.993469   7.0057     3.00126   -0.996869
  3.0008    3.00601   -0.992398  -6.99154   -4.99213
  4.00897  -2.99841    1.00985    3.0081     0.00310363
```

```
In [3]: U,σ,V=svd(A)
        UB,μ,VB=svd(B)
```

```
Out[3]: (
8x5 Array{Float64,2}:
  0.463878   -0.10798    -0.382581   -0.291121    0.317776
  0.427491   -0.181731    0.431732   -0.360747   -0.263873
  0.414048    0.710101    0.216101    0.124978   -0.323277
 -0.566599    0.49832     0.170319   -0.313457    0.154328
 -0.151043   -0.0224838  -0.133424   -0.713355   -0.0720899
 -0.248715   -0.284868   -0.0878237  -0.0366098  -0.789693
  0.0501945   0.297962   -0.751785    0.0520642  -0.264282
 -0.149686   -0.179754    0.0288935   0.39788     0.0490616,

[23.296723985600188,16.16869058552629,10.848397890709466,9.528613533804307,4.619678
5x5 Array{Float64,2}:
 -0.313136    0.0187287  -0.298524    0.829526   -0.352654
  0.654249    0.01248    -0.165929   -0.122319   -0.727535
 -0.669695   -0.224843   -0.0603209  -0.486479   -0.510544
  0.13511    -0.903602    0.356012    0.196054   -0.00815807
 -0.0846072   0.363926    0.867737    0.147734   -0.292585   )
```

```
In [4]: # Mirsky's Theorems
        maxabs(σ-μ), norm(ΔA), vecnorm(σ-μ), vecnorm(ΔA)
```

```
Out[4]: (0.013372622807068524,0.0359353499241745,0.016621804157901463,0.03976948089570283)
```

```
In [5]: # Residual bounds - how close is (x,ζ,y) to (U[:,j],σ[j],V[:,j])
        j=rand(2:k-1)
        x=round(U[:,j],3)
        y=round(V[:,j],3)
        x=x/norm(x)
        y=y/norm(y)
        ζ=(x'*A*y)[]
        σ, j, ζ
```

```
Out[5]: ([23.292468466143806,16.172273554647937,10.856101762031024,9.531289460898613,4.6063
```

```
In [6]: # Fact 2
        r=A*y-ζ*x
        s=A'*x-ζ*y
        ϵ=max(norm(r),norm(s))
```

```
Out[6]: 0.009236418624467026

In [7]: minimum(abs(σ-ζ)), ϵ

Out[7]: (4.246295564058755e-7,0.009236418624467026)

In [8]: # Fact 4
        η=min(abs(ζ-σ[j-1]),abs(ζ-σ[j+1]))

Out[8]: 1.3248118765028547

In [9]: ζ-σ[j], ϵ^2/η

Out[9]: (4.246295564058755e-7,6.439512697576388e-5)

In [10]: # Eigenvector bound
         # cos(θ)
         cosθU=dot(x,U[:,j])
         cosθV=dot(y,V[:,j])
         # Bound
         sqrt(1-cosθU^2+1-cosθV^2), sqrt(norm(r)^2+norm(s)^2)/η

Out[10]: (0.0006863789507620033,0.007841809791517083)

In [11]: # Fact 5 - we create small off-diagonal block perturbation
         j=3
         M=A[1:j,1:j]
         H=A[j+1:m,j+1:n]
         B=cat([1,2],M,H)
         E=rand(size(A[1:j,j+1:n]))/100
         F=rand(size(A[j+1:m,1:j]))/100
         C=map(Float64,B)
         C[1:j,j+1:n]=E
         C[j+1:m,1:j]=F
         C

Out[11]: 8x5 Array{Float64,2}:
          -5.0         7.0          -6.0         0.00535031   0.00029281
          -7.0         7.0          -4.0         0.00146746   0.00745401
          -2.0         7.0          -9.0         0.00131762   0.0034951
           0.00353067  0.000344556   0.00336747  -9.0          5.0
           0.00575373  0.00183245    0.00666969  -2.0         -2.0
           0.00128292  0.00936926    0.00168663   3.0         -1.0
           0.00692802  0.0036979     0.00874577  -7.0         -5.0
           0.00597927  0.00733959    0.00362242   3.0          0.0

In [12]: ϵ=max(norm(E), norm(F))
         β=svdvals(B)
         γ=svdvals(C)
         η=minabs(svdvals(M).-svdvals(H)')
         [β γ], maxabs(β-γ), 2*ϵ^2/(η+sqrt(η^2+4*ϵ^2))
```

```
Out[12]: (
         5x2 Array{Float64,2}:
          18.2486       18.2486
          12.3624       12.3624
           7.36016       7.36017
           4.99903       4.99903
           7.36571e-16   0.0157183,

         0.015718303207464056,0.0001513156165687041)
```

7.4 Relative perturbation theory

7.4.1 Definitions

Matrix $A \in \mathbb{C}^{m\times n}$ is **multiplicatively pertubed** to $\tilde A$ if $\tilde A = D_L^* A D_R$ for some $D_L \in \mathbb{C}^{m\times m}$ and $D_R \in \mathbb{C}^{n\times n}$.

Matrix A is (highly) **graded** if it can be scaled as $A = GS$ such that G is *well-behaved* (that is, $\kappa_2(G)$ is of modest magnitude), where the **scaling matrix** S is often diagonal. Interesting cases are when $\kappa_2(G) \ll \kappa_2(A)$.

Relative distances between two complex numbers α and $\tilde\alpha$ are:

$$
\begin{aligned}
\zeta(\alpha,\tilde\alpha) &= \frac{|\alpha-\tilde\alpha|}{\sqrt{|\alpha\tilde\alpha|}}, \quad \text{for } \alpha\tilde\alpha \neq 0,\\
\varrho(\alpha,\tilde\alpha) &= \frac{|\alpha-\tilde\alpha|}{\sqrt{|\alpha|^2+|\tilde\alpha|^2}}, \quad \text{for } |\alpha|+|\tilde\alpha| > 0.
\end{aligned}
$$

7.4.2 Facts

1. If D_L and D_R are non-singular and $m \geq n$, then

$$
\begin{aligned}
\frac{\sigma_j}{\|D_L^{-1}\|_2\|D_R^{-1}\|_2} \leq \tilde\sigma_j \leq \sigma_j \|D_L\|_2\|D_R\|_2, \quad \text{for } i=1,\ldots,n,\\
\|\operatorname{diag}(\zeta(\sigma_1,\tilde\sigma_1),\ldots,\zeta(\sigma_n,\tilde\sigma_n)\|_{2,F} \leq \frac{1}{2}\|D_L^*-D_L^{-1}\|_{2,F} + \frac{1}{2}\|D_R^*-D_R^{-1}\|_{2,F}.
\end{aligned}
$$

2. Let $m \geq n$ and let

$$
\begin{bmatrix} U_1^* \\ U_2^* \end{bmatrix} A \begin{bmatrix} V_1 & V_2 \end{bmatrix} = \begin{bmatrix} A_1 & \\ & A_2 \end{bmatrix}, \quad \begin{bmatrix} \tilde U_1^* \\ \tilde U_2^* \end{bmatrix} \tilde A \begin{bmatrix} \tilde V_1 & \tilde V_2 \end{bmatrix} = \begin{bmatrix} \tilde A_1 & \\ & \tilde A_2 \end{bmatrix},
$$

where $\begin{bmatrix} U_1 & U_2 \end{bmatrix}$, $\begin{bmatrix} V_1 & V_2 \end{bmatrix}$, $\begin{bmatrix} \tilde U_1 & \tilde U_2 \end{bmatrix}$, and $\begin{bmatrix} \tilde V_1 & \tilde V_2 \end{bmatrix}$ are unitary, and $U_1, \tilde U_1 \in \mathbb{C}^{m\times k}$, $V_1, \tilde V_1 \in \mathbb{C}^{n\times k}$. Set

$$
R = A\tilde V_1 - \tilde U_1 \tilde A_1, \quad S = A^*\tilde U_1 - \tilde V_1 \tilde A_1.
$$

Let $\eta = \min \varrho(\mu,\tilde\mu)$ over all $\mu \in sv(A_1)$ and $\tilde\mu \in sv_{ext}(A_2)$. If $\eta > 0$, then

$$
\begin{aligned}
&\sqrt{\|\sin\Theta(U_1,\tilde U_1)\|_F^2 + \|\sin\Theta(V_1,\tilde V_1)\|_F^2}\\
&\leq \frac{1}{\eta}(\|(I-D_L^*)U_1\|_F^2 + \|(I-D_L^{-1})U_1\|_F^2\\
&\quad + \|(I-D_R^*)V_1\|_F^2 + \|(I-D_R^{-1})V_1\|_F^2)^{1/2}.
\end{aligned}
$$

3. Let $A=GS$ and $\tilde A=\tilde G S$, where G $=n$, and let $\Delta G=\tilde G-G$. Then $\tilde A=DA$, where $D=I+(\Delta G)G^\dagger$, and Fact 1 applies with $D_{=}D$, $D_R=I$, and

$$\|D^*-D^{-1}\|_{2,F}\leq \left(1+\frac{1}{1-\|(\Delta G)G^\dagger\|_2}\right)\frac{\|(\Delta G)G^\dagger\|_{2,F}}{2}.$$

   According to the notebook on Jacobi Method and High Relative Accuracy, nearly optimal diagonal scaling is such that all columns of G have unit norms, $S=\operatorname{diag}\big(\|A_{:,1}\|_2,\ldots,\|A_{:,n}\|_2\big)$.

4. Let A be an real upper-bidiagonal matrix with diagonal entries $a_1,a_2,\ldots,a_n$ and the super-diagonal entries $b_1,b_2,\ldots,b_{n-1}$. Let the diagonal entries of $\tilde A$ be $\alpha_1 a_1,\alpha_2 a_2,\ldots,\alpha_n a_n$, and its super-diagonal entries be $\beta_1 b_1,\beta_2 b_2,\ldots,\beta_{n-1}b_{n-1}$. Then $\tilde A=D_L^*AD_R$ with

$$\begin{aligned}
D_L&=\operatorname{diag}\left(\alpha_1,\frac{\alpha_1\alpha_2}{\beta_1},\frac{\alpha_1\alpha_2\alpha_3}{\beta_1\beta_2},\cdots\right),\\
D_R&=\operatorname{diag}\left(1,\frac{\beta_1}{\alpha_1},\frac{\beta_1\beta_2}{\alpha_1\alpha_2},\cdots\right).
\end{aligned}$$

   Let $\alpha=\prod\limits_{j=1}^{n}\max\{\alpha_j,1/\alpha_j\}$ and $\beta=\prod\limits_{j=1}^{n-1}\max\{\beta_j,1/\beta_j\}$. Then

$$(\alpha\beta)^{-1}\leq \big(\|D_L^{-1}\|_2\|D_R^{-1}\|_2\leq \|D_L\|_2\|D_R\|_2\leq \alpha\beta,$$

   and Fact 1 applies.

5. Consider the block partitioned matrices

$$\begin{aligned}
A&=\begin{bmatrix} B & C\\ 0 & D\end{bmatrix},\\
\tilde A&=\begin{bmatrix} B & 0\\ 0 & D\end{bmatrix}=A\begin{bmatrix} I & -B^{-1}C\\ 0 & I\end{bmatrix}\equiv AD_R.
\end{aligned}$$

   By Fact 1, $\zeta(\sigma_j,\tilde\sigma_j)\leq \frac{1}{2}\|B^{-1}C\|_2$. This is used as a deflation criterion in the SVD algorithm for bidiagonal matrices.

7.4.3 Example - Bidiagonal matrix

In oreder to illustrate Facts 1 to 3, we need an algorithm which comutes the singulačr valoues with high relative acuracy. Such algorithm, the one-sided Jacobi method, is discussed in the following notebook.

The algorithm actually used in the function `svdvals()` for `Bidiagonal` is the zero-shift bidiagonal QR algorithm, which attains the accuracy given by Fact 4: if all $1-\varepsilon\leq\alpha_i,\beta_j\leq 1+\varepsilon$, then

$$(1-\varepsilon)^{2n-1}\leq(\alpha\beta)^{-1}\leq\alpha\beta\leq(1-\varepsilon)^{2n-1}.$$

In other words, ε relative changes in diagonal and super-diagonal elements, cause at most $(2n-1)\varepsilon$ relative changes in the singular values.

However, if singular values and vectors are desired, the function `svd()` calls the standard algorithm, described in the next notebook, which **does not attain this accuracy** .

```
In [13]: n=50
         δ=100000
         # The starting matrix
         a=exp(50*(rand(n)-0.5))
         b=exp(50*(rand(n-1)-0.5))
         A=Bidiagonal(a,b, true)
         # Multiplicative perturbation
         DL=ones(n)+(rand(n)-0.5)/δ
         DR=ones(n)+(rand(n)-0.5)/δ
         # The perturbed matrix
         α=DL.*a.*DR
         β=DL[1:end-1].*b.*DR[2:end]
         B=Bidiagonal(α,β,true)

Out[13]: 50x50 Bidiagonal{Float64}:
          2.08477e-9  3979.12      ...     0.0         0.0           0.0
          0.0            0.0160994       0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0         ...     0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0         ...     0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          ⋮                          ⋱
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0         ...     0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0             0.0         0.0           0.0
          0.0            0.0         ...     0.0         0.0           0.0
          0.0            0.0          7549.47        0.0           0.0
          0.0            0.0             3.9053e9    0.00615868    0.0
          0.0            0.0             0.0         7.31699e-6    2.01869e9
          0.0            0.0             0.0         0.0           6.34784e-7

In [14]: (a-α)./a, (b-β)./b

Out[14]: ([4.10308e-6,-7.74756e-6,-4.14313e-7,1.26467e-6,2.91084e-6,5.7071e-7,-1.2077e-6,7.2

In [15]: @which svdvals(A)

Out[15]: svdvals{T<:Union{Complex{Float32},Complex{Float64},Float32,Float64}}(A::AbstractArr

In [16]: σ=svdvals(A)
         μ=svdvals(B)
         [σ (σ-μ)./σ]
```

```
Out[16]: 50x2 Array{Float64,2}:
          6.62726e10     -7.17766e-6
          5.10335e10      1.15467e-6
          4.36214e10     -3.80057e-6
          4.05512e10     -8.70757e-6
          3.44475e10     -7.13151e-6
          1.53257e10     -2.0363e-6
          3.9942e9        2.08668e-6
          3.9053e9       -1.87447e-7
          3.89047e9       5.12241e-6
          2.73723e9       2.40934e-7
          2.19498e9       4.81754e-6
          2.01868e9      -2.26216e-6
          1.20977e9      -1.25286e-6
          ⋮
          0.0030481      -4.90262e-6
          0.000173653     6.43197e-7
          8.3336e-5      -6.49589e-6
          9.51231e-6     -3.95159e-6
          4.93278e-6      6.53628e-6
          1.19055e-8     -1.39709e-6
          8.16637e-9     -9.5009e-6
          2.58052e-11     5.59597e-6
          8.06225e-12     5.24998e-6
          1.96207e-15     3.34629e-6
          1.07145e-18     2.05903e-6
          6.10009e-122    1.50271e-6

In [17]: cond(A)

Out[17]: 1.0864191503808956e132

In [18]: # The standard algorithm
         U,ν,V=svd(A);

In [19]: (σ-ν)./σ

Out[19]: 50-element Array{Float64,1}:
          -1.15121e-16
          -1.49498e-16
           1.749e-16
          -1.88142e-16
           0.0
          -1.24454e-16
           0.0
           0.0
           0.0
          -1.74204e-16
           0.0
           0.0
           0.0
```

```
   ⋮
 -2.75535e-7
  0.0
  0.0
 -1.84806e-12
 -0.342439
  1.38957e-16
  0.999008
  0.70221
  0.997497
  0.999982
  1.0
 -5.0563e60
```

```
In [ ]:
```

8 Singular Value Decomposition - Algorithms and Error Analysis

---

We study only algorithms for real matrices, which are most commonly used in the applications described in this course.
For more details, see A. Kaylor Cline and I. Dhillon, Computation of the Singular Value Decomposition and the references therein.

8.1 Prerequisites

The reader should be familiar with facts about the singular value decomposition and perturbation theory and algorithms for the symmetric eigenvalue decomposition.

8.2 Competences

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

8.3 Basics

8.3.1 Definitions

The **signular value decomposition** (SVD) of $A \in \mathbb{R}^{m\times n}$ is $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m\times m}$ is orthogonal, $U^TU = UU^T = I_m$, $V \in \mathbb{R}^{n\times n}$ is orthogonal, $V^TV = VV^T = I_n$, and $\Sigma \in \mathbb{R}^{m\times n}$ is diagonal with singular values $\sigma_1, \ldots, \sigma_{\min\{m,n\}}$ on the diagonal.
If $m > n$, the **thin SVD** of A is $A = U_{1:m,1:n}\Sigma_{1:n,1:n}V^T$.

8.3.2 Facts

1. Algorithms for computing SVD of A are modifications of algorithms for the symmetric eigenvalue decomposition of the matrices AA^T, A^TA and $\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$.

2. Most commonly used approach is the three-step algorithm:

    1. Reduce A to bidiagonal matrix B by orthogonal transformations, $X^TAY = B$.
    2. Compute the SVD of B with QR iterations, $B = W\Sigma Z^T$.
    3. Multiply $U = XW$ and $V = YZ$.

3. If $m \geq n$, the overall operation count for this algorithm is $O(mn^2)$ operations.

4. **Error bounds**: Let $U\Sigma V^T$ and $\tilde{U}\tilde{\Sigma}\tilde{V}^T$ be the exact and the computed SVDs of A, respectively. The algorithms generally compute the SVD with errors bounded by

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

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 $O(n^2)$). These bounds are obtained by combining perturbation bounds with the floating-point error analysis of the algorithms.

8.4 Bidiagonalization

8.4.1 Facts

1. The reduction of A to bidiagonal matrix can be performed by applying $\min\{m-1,n\}$ Householder reflections H_L from the left and $n-2$ Householder reflections H_R from the right. In the first step, H_L is chosen to annihilate all elements of the first column below the diagonal, and H_R is chosen to annihilate all elements of the first row right of the first super-diagonal. Applying this procedure recursively yields the bidiagonal matrix.

2. H_L and H_R do not depend on the normalization of the respective Householder vectors v_L and v_R. With the normalization $[v_L]_1 = [V_R]_1 = 1$, the vectors v_L are stored in the lower-triangular part of A, and the vectors v_R are stored in the upper-triangular part of A above the super-diagonal.

3. The matrices H_L and H_R are not formed explicitly - given v_L and v_R, A is overwritten with $H_L A H_R$ in $O(mn)$ operations by using matrix-vector multiplication and rank-one updates.

4. Instead of performing rank-one updates, p transformations can be accumulated, and then applied. 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.

5. If the matrices X and Y are needed explicitly, they can be computed from the stored Householder vectors. In order to minimize the operation count, the computation starts from the smallest matrix and the size is gradually increased.

6. The backward error bounds for the bidiagonalization are as follows: The computed matrix $\tilde{B}$ is equal to the matrix which would be obtained by exact bidiagonalization 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 matrices $\tilde{X}$ and $\tilde{Y}$ satisfy $\tilde{X} = X + \Delta X$ and $\tilde{Y} = Y + \Delta Y$, where $\|\Delta X\|_2, \|\Delta Y\|_2 \leq \phi\varepsilon$ and ϕ is a slowly increasing function of n.

7. The bidiagonal reduction is implemented in the LAPACK subroutine DGEBRD. The computation of X and Y is implemented in the subroutine DORGBR, which is not yet wrapped in Julia.

8. Bidiagonalization can also be performed using Givens rotations. 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. Error bounds for function `myBidiagG()` are the same as above, but with slightly different functions ψ and ϕ.

```
In [1]: m=8
        n=5
        A=map(Float64,rand(-9:9,m,n))
```

```
Out[1]: 8x5 Array{Float64,2}:
         -9.0  -6.0  -9.0  -8.0  -9.0
          1.0  -4.0  -9.0   2.0   8.0
          2.0  -3.0   2.0   6.0  -2.0
          3.0   9.0   3.0  -9.0  -6.0
          7.0   6.0  -1.0  -6.0  -8.0
          9.0   1.0  -1.0   5.0   0.0
         -6.0   2.0   5.0   4.0  -5.0
          9.0  -2.0   7.0   0.0  -7.0
```

```
In [2]: ?LAPACK.gebrd!
```

```
Out[2]:
```

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce `A` in-place to bidiagonal form `A = QBP'`. Returns `A`, containing the bidiagonal matrix `B`; `d`, containing the diagonal elements of `B`; `e`, containing the off-diagonal elements of `B`; `tauq`, containing the elementary reflectors representing `Q`; and `taup`, containing the elementary reflectors representing `P`.

```
In [3]: # We need copy()
        Out=LAPACK.gebrd!(copy(A))

Out[3]: (
        8x5 Array{Float64,2}:
         18.4932       -7.79789       0.431822    0.160875    -0.0931382
         -0.0363726  -16.4195        -5.55048    -0.334302    -0.432763
         -0.0727451   -0.000921678  16.7671     -11.119        0.48751
         -0.109118    -0.163374       0.335752  -11.0792      -7.12552
         -0.254608     0.0751739      0.544518    0.400533     9.13753
         -0.327353     0.182766       0.10458     0.107658     0.491441
          0.218235    -0.401176      -0.306529    0.00780384   0.536132
         -0.327353     0.0519265      0.671579   -0.271868    -0.355809 ,

        [18.49324200890693,-16.419500108180916,16.767114699176663,-11.079206500900566,9.1375

In [4]: B=Bidiagonal(Out[2],Out[3][1:end-1],true)

Out[4]: 5x5 Bidiagonal{Float64}:
         18.4932   -7.79789   0.0        0.0       0.0
          0.0     -16.4195   -5.55048    0.0       0.0
          0.0       0.0      16.7671   -11.119     0.0
          0.0       0.0       0.0      -11.0792   -7.12552
          0.0       0.0       0.0        0.0       9.13753

In [5]: svdvals(A), svdvals(B)

Out[5]: ([23.05690050927775,20.9332244124872,14.552987762575267,12.111135956464679,6.058909

In [6]: # Extract X
        function myBidiagX{T}(H::Array{T})
            m,n=size(H)
            X = eye(T,m,n)
            v=Array(T,m)
            for j = n : -1 : 1
                v[j] = one(T)
                v[j+1:m] = H[j+1:m, j]
                γ = -2 / (v[j:m]⋅v[j:m])
                w = γ * X[j:m, j:n]'*v[j:m]
                X[j:m, j:n] = X[j:m, j:n] + v[j:m]*w'
            end
```

```
        X
    end

    # Extract Y
    function myBidiagY{T}(H::Array{T})
        n,m=size(H)
        Y = eye(T,n)
        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 * Y[j+1:n, j+1:n]'*v[j+1:n]
            Y[j+1:n, j+1:n] = Y[j+1:n, j+1:n] + v[j+1:n]*w'
        end
        Y
    end
```

```
Out[6]: myBidiagY (generic function with 1 method)
```

```
In [7]: X=myBidiagX(Out[1])
```

```
Out[7]: 8x5 Array{Float64,2}:
 -0.486664   -0.434465   -0.671704   -0.0178347  -0.269437
  0.0540738  -0.611028    0.294346   -0.148881   -0.0275076
  0.108148    0.0331046   0.0308777   0.280724   -0.464869
  0.162221    0.31319    -0.31251    -0.485002    0.304868
  0.378517   -0.011677   -0.362864   -0.497537   -0.233032
  0.486664   -0.155106    0.16278    -0.16509    -0.516462
 -0.324443    0.557829    0.0570937  -0.0420409  -0.539626
  0.486664    0.0577478  -0.44959     0.622028    0.0732598
```

```
In [8]: Y=myBidiagY(Out[1]')
```

```
Out[8]: 5x5 Array{Float64,2}:
 1.0   0.0        0.0        0.0        0.0
 0.0  -0.637967   0.347683   0.619034   0.298182
 0.0  -0.707311  -0.389459  -0.574597   0.133685
 0.0  -0.263508   0.570624  -0.234905  -0.741466
 0.0   0.152557   0.633898  -0.481098   0.586041
```

```
In [9]: # Orthogonality
        norm(X'*X-I), norm(Y'*Y-I)
```

```
Out[9]: (6.081307256636832e-16,2.647317538811372e-16)
```

```
In [10]: # Error
         X'*A*Y-B
```

```
Out[10]: 5x5 Array{Float64,2}:
  3.55271e-15  -2.66454e-15   5.55112e-16  -1.88738e-15  -2.22045e-16
```

```
 1.11022e-16  -3.55271e-15  -8.88178e-16  -8.88178e-16  -1.77636e-15
 3.55271e-15  -1.33227e-15   0.0          -3.55271e-15  -1.77636e-15
-8.88178e-16  -1.05471e-15  -2.55351e-15  -1.77636e-15  -3.55271e-15
 9.99201e-16  -1.11022e-15   1.77636e-15   0.0           0.0
```

In [11]:
```
# Bidiagonalization using Givens rotations
function myBidiagG{T}(A::Array{T})
    m,n=size(A)
    X=eye(T,m,m)
    Y=eye(T,n,n)
    for j = 1 : n
        for i = j+1 : m
            G,r=givens(A,j,i,j)
            A=G*A
            X=G*X
        end
        for i=j+2:n
            G,r=givens(A',j+1,i,j)
            A=A*G'
            Y*=G'
        end
    end
    X',Bidiagonal(diag(A),diag(A,1),true), Y
end
```

Out[11]:
```
myBidiagG (generic function with 1 method)
```

In [12]:
```
X1, B1, Y1 = myBidiagG(A)
```

Out[12]:
```
(
8x8 Array{Float64,2}:
  0.486664     0.434465    -0.671704   ...  -0.0259927  -0.221603   -0.0226884
 -0.0540738    0.611028     0.294346       -0.362203    0.215523    0.580102
 -0.108148    -0.0331046    0.0308777      -0.648687    0.268326   -0.445592
 -0.162221    -0.31319     -0.31251        -0.593123   -0.295628    0.102816
 -0.378517     0.011677    -0.362864        0.274054    0.587808   -0.0494786
 -0.486664     0.155106     0.16278    ...   0.14296    -0.63005    -0.0348522
  0.324443    -0.557829     0.0570937       0.0         0.0506539   0.533643
 -0.486664    -0.0577478   -0.44959         0.0         0.0         0.406702 ,

5x5 Bidiagonal{Float64}:
 diag: -18.4932  16.4195  16.7671  11.0792  9.13753
 super: 7.79789  5.55048  11.119  -7.12552,

5x5 Array{Float64,2}:
 1.0   0.0        0.0        0.0        0.0
 0.0  -0.637967   0.347683  -0.619034   0.298182
 0.0  -0.707311  -0.389459   0.574597   0.133685
 0.0  -0.263508   0.570624   0.234905  -0.741466
 0.0   0.152557   0.633898   0.481098   0.586041)
```

In [13]:
```
# Orthogonality
norm(X1'*X1-I), norm(Y1'*Y1-I)
```

```
Out[13]: (1.032766086574982e-15,6.521546438536692e-16)

In [14]: # Error
         X1'*A*Y1

Out[14]: 8x5 Array{Float64,2}:
          -18.4932          7.79789       1.22125e-15   2.44249e-15  -8.88178e-16
           -7.77156e-16   16.4195         5.55048       1.33227e-15  -2.66454e-15
           -8.88178e-16    1.33227e-15   16.7671       11.119         0.0
           -8.88178e-16    7.77156e-16   -2.22045e-16  11.0792       -7.12552
           -2.22045e-16   -1.9984e-15    -8.88178e-16  -8.88178e-16   9.13753
            2.22045e-16    7.52168e-16    8.28926e-16   1.8577e-16    1.02841e-15
            1.22125e-15    9.68503e-16    3.54737e-16  -1.37317e-15   1.78392e-17
           -1.77636e-15    2.80235e-16    9.57071e-16  -6.21728e-17   1.2162e-15

In [15]: # X, Y and B are not unique
         B

Out[15]: 5x5 Bidiagonal{Float64}:
          18.4932   -7.79789    0.0         0.0       0.0
           0.0     -16.4195    -5.55048     0.0       0.0
           0.0        0.0      16.7671    -11.119     0.0
           0.0        0.0       0.0       -11.0792   -7.12552
           0.0        0.0       0.0         0.0       9.13753
```

8.5 Bidiagonal QR method

Let B be a real upper-bidiagonal matrix of order n and let $B = W\Sigma Z^T$ be its SVD.
All metods for computing the SVD of bidiagonal matrix are derived from the methods for computing the EVD of the tridiagonal matrix $T = B^T B$.

8.5.1 Facts

1. The shift μ is the eigenvalue of the 2×2 matrix $T_{n-1:n,n-1:n}$ which is closer to $T_{n,n}$. The first Givens rotation from the right is the one which annihilates the element $(1,2)$ of the shifted 2×2 matrix $T_{1:2,1:2} - \mu I$. Applying this rotation to B creates the bulge at the element $B_{2,1}$. This bulge is subsequently chased out by applying adequate Givens rotations alternating from the left and from the right. This is the **Golub-Kahan algorithm**.

2. The computed SVD satisfes error bounds from the Fact 4 above.

3. The special variant of zero-shift QR algorithm (the **Demmel-Kahan algorithm**) computes the singular values with high relative accuracy.

4. The tridiagonal divide-and-conquer method, bisection and inverse iteration, and MRRR method can also be adapted for bidiagonal matrices.

5. Zero shift QR algorithm for bidiagonal matrices is implemented in the LAPACK routine DBDSQR. It is also used in the function `svdvals()`. Divide-and-conquer algorithm for bidiagonal matrices is implemented in the LAPACK routine DBDSDC. However, this algorithm also calls zero-shift QR to compute singular values.

8.5.2 Examples

```
In [16]: W,σ,Z=svd(B)

Out[16]: (
         5x5 Array{Float64,2}:
           0.539491   -0.668158   -0.464621   0.213637   0.0316369
           0.542761   -0.184696    0.693163  -0.427363  -0.0904592
          -0.606223   -0.645776    0.020825  -0.412012  -0.212787
          -0.214206   -0.31478     0.491036   0.540329   0.567412
           0.0311227   0.0577809  -0.249201  -0.556786   0.789672 ,

         [23.05690050927775,20.933224412487217,14.552987762575265,12.111135956464674,6.0589
         5x5 Array{Float64,2}:
           0.432709   -0.590277  -0.590419   0.326216   0.0965633
          -0.568973    0.393768  -0.533109   0.441838   0.204425
          -0.571508   -0.468282  -0.240378  -0.374547  -0.505988
           0.395277    0.509617  -0.389738  -0.116029  -0.647062
           0.0785325   0.132371  -0.396892  -0.73798    0.523615 )

In [17]: @which svd(B)

Out[17]: svd{T<:Union{Float32,Float64}}(M::Bidiagonal{T<:Union{Float32,Float64}}) at linalg/

In [18]: σ1=svdvals(B)

Out[18]: 5-element Array{Float64,1}:
          23.0569
          20.9332
          14.553
          12.1111
           6.05891

In [19]: @which svdvals(B)

Out[19]: svdvals{T<:Union{Complex{Float32},Complex{Float64},Float32,Float64}}(A::AbstractArr

In [20]: σ-σ1

Out[20]: 5-element Array{Float64,1}:
           0.0
           1.77636e-14
          -1.77636e-15
          -5.32907e-15
          -8.88178e-16

In [23]: ?LAPACK.bdsqr!

Out[23]:

bdsqr!(uplo, d, e, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e` on the off-diagonal. If `uplo = U`, `e` is the superdiagonal. If `uplo = L`, `e` is the subdiagonal. Can optionally also compute the product `Q' * C`. Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

```
In [24]: BV=eye(n)
         BU=eye(n)
         BC=eye(n)
         σ2,Z2,W2,C = LAPACK.bdsqr!('U',copy(B.dv),copy(B.ev),BV,BU,BC)

Out[24]: ([23.05690050927775,20.933224412487217,14.552987762575265,12.111135956464674,6.0589
         5x5 Array{Float64,2}:
           0.432709   -0.568973  -0.571508   0.395277   0.0785325
          -0.590277    0.393768  -0.468282   0.509617   0.132371
          -0.590419   -0.533109  -0.240378  -0.389738  -0.396892
           0.326216    0.441838  -0.374547  -0.116029  -0.73798
           0.0965633   0.204425  -0.505988  -0.647062   0.523615 ,

         5x5 Array{Float64,2}:
           0.539491   -0.668158   -0.464621   0.213637   0.0316369
           0.542761   -0.184696    0.693163  -0.427363  -0.0904592
          -0.606223   -0.645776    0.020825  -0.412012  -0.212787
          -0.214206   -0.31478     0.491036   0.540329   0.567412
           0.0311227   0.0577809  -0.249201  -0.556786   0.789672 ,

         5x5 Array{Float64,2}:
           0.539491    0.542761   -0.606223  -0.214206   0.0311227
          -0.668158   -0.184696   -0.645776  -0.31478    0.0577809
          -0.464621    0.693163    0.020825   0.491036  -0.249201
           0.213637   -0.427363   -0.412012   0.540329  -0.556786
           0.0316369  -0.0904592  -0.212787   0.567412   0.789672 )

In [25]: W2'*full(B)*Z2'

Out[25]: 5x5 Array{Float64,2}:
          23.0569       -1.33227e-15   3.10862e-15  -2.22045e-15  -1.11022e-16
          -6.7446e-15   20.9332       -6.66134e-16  -4.44089e-16   0.0
           5.55112e-15   3.10862e-15  14.553         0.0           2.22045e-15
          -1.11022e-15   2.22045e-16   2.22045e-15  12.1111        1.86517e-14
          -1.13798e-15  -8.32667e-16   0.0           0.0           6.05891

In [26]: ?LAPACK.bdsdc!

Out[26]:
```

`bdsdc!(uplo, compq, d, e) -> (d, e, u, vt, q, iq)`

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e` on the off-diagonal using a divide and conqueq method. If `uplo = U`, `e` is the superdiagonal. If `uplo = L`, `e` is the subdiagonal. If `compq = N`, only the singular values are found. If `compq = I`, the singular values and vectors are found. If `compq = P`, the singular values and vectors are found in compact form. Only works for real types. Returns the singular values in `d`, and if `compq = P`, the compact singular vectors in `iq`.

```
In [27]: σ3,ee,W3,Z3,rest=LAPACK.bdsdc!('U','I',copy(B.dv),copy(B.ev))

Out[27]: ([23.05690050927775,20.933224412487217,14.552987762575265,12.111135956464674,6.058
         5x5 Array{Float64,2}:
           0.539491   -0.668158   -0.464621   0.213637   0.0316369
           0.542761   -0.184696    0.693163  -0.427363  -0.0904592
          -0.606223   -0.645776    0.020825  -0.412012  -0.212787
          -0.214206   -0.31478     0.491036   0.540329   0.567412
           0.0311227   0.0577809  -0.249201  -0.556786   0.789672 ,

         5x5 Array{Float64,2}:
           0.432709   -0.568973  -0.571508   0.395277   0.0785325
          -0.590277    0.393768  -0.468282   0.509617   0.132371
          -0.590419   -0.533109  -0.240378  -0.389738  -0.396892
           0.326216    0.441838  -0.374547  -0.116029  -0.73798
           0.0965633   0.204425  -0.505988  -0.647062   0.523615 ,

         [6.9130615359347e-310],[139921923212720])

In [28]: W3'*full(B)*Z3'

Out[28]: 5x5 Array{Float64,2}:
          23.0569       -1.33227e-15   3.10862e-15  -2.22045e-15  -1.11022e-16
          -6.7446e-15   20.9332        -6.66134e-16  -4.44089e-16   0.0
           5.55112e-15   3.10862e-15  14.553          0.0           2.22045e-15
          -1.11022e-15   2.22045e-16   2.22045e-15  12.1111         1.86517e-14
          -1.13798e-15  -8.32667e-16   0.0           0.0           6.05891
```

Functions `svd()`, `LAPACK.bdsqr!()` and `LAPACK.bdsdc!()` use the same algorithm to compute singular values.

```
In [29]: [σ3-σ2 σ3-σ]

Out[29]: 5x2 Array{Float64,2}:
          0.0  0.0
          0.0  0.0
          0.0  0.0
          0.0  0.0
          0.0  0.0
```

Let us compute some timings. We observe $O(n^2)$ operations.

```
In [30]: n=1000
         Abig=Bidiagonal(rand(n),rand(n-1),true)
         Bbig=Bidiagonal(rand(2*n),rand(2*n-1),true)
         @time svdvals(Abig);
         @time svdvals(Bbig);
         @time LAPACK.bdsdc!('U','N',copy(Abig.dv),copy(Abig.ev));
         @time svd(Abig);
         @time svd(Bbig);
```

```
0.036636 seconds (168 allocations: 151.667 KB)
0.086415 seconds (24 allocations: 282.297 KB)
0.025992 seconds (23 allocations: 141.641 KB)
0.151774 seconds (33 allocations: 45.884 MB, 3.92% gc time)
0.543586 seconds (33 allocations: 183.320 MB, 2.35% gc time)
```

8.6 QR method

Final algorithm is obtained by combining bidiagonalization and bidiagonal SVD methods. Standard method is implemented in the LAPACK routine DGESVD. Divide-and-conquer method is implemented in the LAPACK routine DGESDD.

The functions `svd()`, `svdvals()`, and `svdvecs()` use `DGESDD`. Wrappers for `DGESVD` and `DGESDD` give more control about output of eigenvectors.

```
In [31]: # The built-in algorithm
         U,σA,V=svd(A)
```

```
Out[31]: (
         8x5 Array{Float64,2}:
          -0.0957244   0.829229   -0.0306424   0.498836   -0.0560516
          -0.449874   -0.0680817  -0.508787    0.0862792  -0.111848
          -0.0170071  -0.213541    0.227033    0.40675    -0.213951
           0.560334    0.205861   -0.178915   -0.402239    0.00884894
           0.517168    0.126727   -0.377755    0.0962755  -0.376083
           0.098974   -0.379515   -0.282601    0.301544   -0.506719
           0.0853335   0.0589339   0.65243    -0.0534916  -0.522856
           0.435484   -0.237068    0.0917332   0.559837    0.516637   ,

         [23.05690050927775,20.933224412487217,14.552987762575265,12.111135956464674,6.0589
         5x5 Array{Float64,2}:
           0.432709  -0.590277   -0.590419   0.326216   0.0965633
           0.432389  -0.0590841  -0.103076  -0.70398   -0.55076
           0.408393  -0.371268    0.641575  -0.198633   0.494269
          -0.327269  -0.588834    0.389146   0.244289  -0.578842
          -0.593222  -0.404372   -0.278798  -0.546684   0.328602 )
```

```
In [32]: # With our building blocks
         U1=X*W
         V1=Y*Z
         U1'*A*V1
```

```
Out[32]: 5x5 Array{Float64,2}:
          23.0569       -4.44089e-15   3.10862e-15  -4.44089e-15  -8.88178e-16
          -8.88178e-15  20.9332        0.0          -4.44089e-15  -1.33227e-15
           7.54952e-15   3.33067e-15  14.553         2.22045e-15   2.88658e-15
          -8.88178e-16   3.10862e-15   4.44089e-15  12.1111        1.64313e-14
          -2.22045e-16  -9.99201e-16   8.88178e-16   1.9984e-15    6.05891
```

```
In [33]: ?LAPACK.gesvd!
```

```
Out[33]:
```

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`. If `jobu = A`, all the columns of `U` are computed. If `jobvt = A` all the rows of `V'` are computed. If `jobu = N`, no columns of `U` are computed. If `jobvt = N` no rows of `V'` are computed. If `jobu = O`, `A` is overwritten with the columns of (thin) `U`. If `jobvt = O`, `A` is overwritten with the rows of (thin) `V'`. If `jobu = S`, the columns of (thin) `U` are computed and returned separately. If `jobvt = S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `O`. Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

```
In [34]: # DGESVD
         LAPACK.gesvd!('A','A',copy(A))

Out[34]: (
         8x8 Array{Float64,2}:
          -0.0957244  -0.829229   -0.0306424   0.498836   …   0.00637886  -0.000941379
          -0.449874    0.0680817  -0.508787    0.0862792      0.64812     -0.179225
          -0.0170071   0.213541    0.227033    0.40675       -0.219861    -0.789206
           0.560334   -0.205861   -0.178915   -0.402239       0.322832    -0.479953
           0.517168   -0.126727   -0.377755    0.0962755     -0.1893       0.158474
           0.098974    0.379515   -0.282601    0.301544   …  -0.027736     0.209763
           0.0853335  -0.0589339   0.65243    -0.0534916      0.49558      0.167152
           0.435484    0.237068    0.0917332   0.559837       0.380969     0.13275     ,

         [23.05690050927775,20.933224412487206,14.552987762575265,12.11113595646468,6.05890
         5x5 Array{Float64,2}:
           0.432709    0.432389    0.408393  -0.327269  -0.593222
           0.590277    0.0590841   0.371268   0.588834   0.404372
          -0.590419   -0.103076    0.641575   0.389146  -0.278798
           0.326216   -0.70398    -0.198633   0.244289  -0.546684
           0.0965633  -0.55076     0.494269  -0.578842   0.328602)

In [35]: ?LAPACK.gesdd!

Out[35]:
```

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`, using a divide and conquer approach. If `job = A`, all the columns of `U` and the rows of `V'` are computed. If `job = N`, no columns of `U` or rows of `V'` are computed. If `job = O`, `A` is overwritten with the columns of (thin) `U` and the rows of (thin) `V'`. If `job = S`, the columns of (thin) `U` and the rows of (thin) `V'` are computed and returned separately.

```
In [36]: LAPACK.gesdd!('N',copy(A))

Out[36]: (8x0 Array{Float64,2},[23.05690050927775,20.9332244124872,14.552987762575267,12.111
```

Let us perform some timings. We observe $O(n^3)$ operations.

```
In [37]: n=1000
         Abig=rand(n,n)
```

```
Bbig=rand(2*n,2*n)
@time Ubig,σbig,Vbig=svd(Abig);
@time svd(Bbig);
@time LAPACK.gesvd!('A','A',copy(Abig));
@time LAPACK.gesdd!('A',copy(Abig));
@time LAPACK.gesdd!('A',copy(Bbig));
```

```
0.561084 seconds (41 allocations: 53.529 MB, 0.38% gc time)
6.330250 seconds (35 allocations: 213.868 MB, 1.13% gc time)
8.577787 seconds (24 allocations: 23.408 MB)
0.533864 seconds (26 allocations: 45.899 MB)
6.277768 seconds (26 allocations: 183.350 MB, 1.00% gc time)
```

In [38]:
```
# Residual
norm(Abig*Vbig-Ubig*diagm(σbig))
```

Out[38]: 6.275333151332255e-13

In [ ]:

9 Singular Value Decomposition - Jacobi and Lanczos Methods

Since computing the SVD of A can be seen as computing the EVD of the symmetric matrices A^*A, AA^*, or $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$, simple modifications of the corresponding EVD algorithms yield version for comuting the SVD.

For more details on one-sided Jacobi method, see Z. Drmač, Computing Eigenvalues and Singular Values to High Relative Accuracy and the references therein.

9.1 Prerequisites

The reader should be familiar with concepts of singular values and vectors, related perturbation theory, and algorithms, and Jacobi and Lanczos methods for the symmetric eigenvalue decomposition.

9.2 Competences

The reader should be able to recognise matrices which warrant high relative accuracy and to apply Jacobi method to them. The reader should be able to recognise matrices to which Lanczos method can be efficiently applied and do so.

9.3 One-sided Jacobi method

Let $A \in \mathbb{R}^{m \times n}$ with $\operatorname{rank}(A) = n$ (therefore, $m \geq n$) and $A = U\Sigma V^T$ its thin SVD.

9.3.1 Definition

Let $A = BD$, where $D = \operatorname{diag}(\|A_{:,1}\|_2, \ldots, \|A_{:,n}\|_2)$ is a **diagonal scaling** , and B is the **scaled matrix** of A from the right. Then $[B^T B]_{i,i} = 1$.

9.3.2 Facts

1. Let $\tilde{U}$, $\tilde{V}$ and $\tilde{\Sigma}$ be the approximations of U, V and Σ, respectively, computed by a backward stable algorithm as $A + \Delta A = \tilde{U}\tilde{\Sigma}\tilde{V}^T$. Since the orthogonality of $\tilde{U}$ and $\tilde{V}$ cannot be guaranteed, this product in general does not represent and SVD. There exist nearby orthogonal matrices $\hat{U}$ and $\hat{V}$ such that $(I+E_1)(A+\Delta A)(I+E_2) = \hat{U}\tilde{\Sigma}\hat{V}^T$, where departures from orthogonalithy, E_1 and E_2, are small in norm.

2. Standard algorithms compute the singular values with backward error $\|\Delta A\| \leq \phi\varepsilon\|A\|_2$, where ε is machine precision and ϕ is a slowly growing function og n. The best error bound for the singular values is $|\sigma_j - \tilde{\sigma}_j| \leq \|\Delta A\|_2$, and the best relative error bound is
$$\max_j \frac{|\sigma_j - \tilde{\sigma}_j|}{\sigma_j} \leq \frac{\|\Delta A\|_2}{\sigma_j} \leq \phi\varepsilon\kappa_2(A).$$

3. Let $\|[\Delta A]_{:,j}\|_2 \leq \varepsilon\|A_{:,j}\|_2$ for all j. Then $A + \Delta A = (B + \Delta B)D$ and $\|\Delta B\|_F \leq \sqrt{n}\varepsilon$, and
$$\max_j \frac{|\sigma_j - \tilde{\sigma}_j|}{\sigma_j} \leq \frac{\|(\Delta B)B^\dagger\|_2}{\leq} \sqrt{n}\varepsilon\|B^\dagger\|_2.$$
This is Fact 3 from the Relative perturbation theory.

4. It holds
$$\|B^\dagger\| \leq \kappa_2(B) \leq \sqrt{n} \min_{S=\mathrm{diag}} \kappa_2(AS) \leq \sqrt{n}\kappa_2(A).$$
Therefore, numerical algorithm with column-wise small backward error computes singular values more accurately than an algorithm with small norm-wise backward error.

5. In each step, one-sided Jacobi method computes the Jacobi roataion matrix from the pivot submatrix of the current matrix A^TA. Afterwards, A is multiplied with the computed rotation matrix from the right (only two columns are affected). Convergence of the Jacobi method for the symmetric matrix A^TA to a diagonal matrix, implies that the matrix A converges to the matrix AV with orthogonal columns and $V^TV = I$. Then $AV = U\Sigma$, $\Sigma = \mathrm{diag}(\|A_{:,1}\|_2, \ldots, \|A_{:,n}\|_2)$, $U = AV\Sigma^{-1}$, and $A = U\Sigma V^T$ is the SVD of A.

6. One-sided Jacobi method computes the SVD with error bound from Facts 2 and 3, provided that the condition of the intermittent scaled matrices does not grow much. There is overwhelming numerical evidence for this. Alternatively, if A is square, the one-sided Jacobi method can be applied to the transposed matrix $A^T = DB^T$ and the same error bounds apply, but the condition of the scaled matrix (*this time from the left*) does not change. This approach is slower.

7. One-sided Jacobi method can be preconditioned by applying one QR factorization with full pivoting and one QR factorization withour pivoting to A, to obtain faster convergence, without sacrifying accuracy. This method is implemented in the LAPACK routine DGESVJ. *Writing the wrapper for* `DGESVJ` *is a tutorial assignment.*

9.3.3 Example - Standard matrix

```
In [1]: function myJacobiR{T}(A::Array{T})
            m,n=size(A)
            V=eye(T,n,n)
            # Tolerance for rotation
            tol=sqrt(n)*eps(T)
            # Counters
            p=n*(n-1)/2
            sweep=0
            pcurrent=0
            # First criterion is for standard accuracy, second one is for relative accuracy
            # while sweep<30 && vecnorm(A-diagm(diag(A)))>tol
            while sweep<30 && pcurrent<p
                sweep+=1
                # Row-cyclic strategy
                for i = 1 : n-1
                    for j = i+1 : n
                        # Compute the 2 x 2 sumbatrix of A'*A
                        F=A[:,[i,j]]'*A[:,[i,j]]
                        # Check the tolerance - the first criterion is standard,
                        # the second one is for relative accuracy
                        # if A[i,j]!=zero(T)
                        #
                        if abs(F[1,2])>tol*sqrt(F[1,1]*F[2,2])
                            # Compute c and s
                            τ=(F[1,1]-F[2,2])/(2*F[1,2])
                            t=sign(τ)/(abs(τ)+sqrt(1+τ^2))
```

```
                        c=1/sqrt(1+t^2)
                        s=c*t
                        G=LinAlg.Givens(i,j,c,s)
                        A*=G'
                        V*=G'
                        pcurrent=0
                    else
                        pcurrent+=1
                    end
                end
            end
        end
        σ=[vecnorm(A[:,k]) for k=1:n]
        for k=1:n
            A[:,k]=A[:,k]/σ[k]
        end
        A, σ, V
    end
```

Out[1]: myJacobiR (generic function with 1 method)

In [2]:
```
m=8
n=5
A=map(Float64,rand(-9:9,m,n))
```

Out[2]:
```
8x5 Array{Float64,2}:
  1.0  -3.0  -4.0   3.0  -6.0
  5.0  -6.0  -4.0   9.0  -9.0
 -4.0  -8.0   1.0  -6.0   7.0
 -1.0   7.0   6.0   0.0   4.0
 -5.0  -6.0   6.0   2.0   5.0
 -6.0   9.0  -8.0  -8.0  -7.0
 -9.0  -4.0   9.0   2.0  -5.0
 -9.0  -2.0  -5.0   0.0   1.0
```

In [3]:
```
U,σ,V=myJacobiR(A)
```

Out[3]:
```
(
8x5 Array{Float64,2}:
  0.252283    0.262687   -0.231463  -0.0402134   0.0642854
  0.646124    0.382589   -0.166955   0.0785412   0.00356519
 -0.0872773  -0.444092   -0.191349  -0.41244     0.574249
 -0.243701   -0.187418    0.303085   0.318266   -0.453511
  0.149336   -0.487201   -0.159112   0.225274   -0.0103944
 -0.619742    0.455695   -0.419944  -0.194352   -0.0911205
  0.158639   -0.322184   -0.556635  -0.305593   -0.628011
 -0.142552   -0.0587595  -0.527792   0.734102    0.240023   ,

[19.858643222502394,21.40947711151549,15.872078477861853,6.291446115163081,13.105854
5x5 Array{Float64,2}:
  0.347592   0.339557   0.785728   -0.339009   0.177723
 -0.627658   0.354419   0.365788    0.290891  -0.511874
```

```
  0.143582  -0.622357   0.205075   -0.344999  -0.656488
  0.637985   0.0762085  0.0553907   0.708112  -0.287536
 -0.239968  -0.60494    0.451339    0.424275   0.439029)
```

In [4]:
```
# Residual
A*V-U*diagm(σ)
```

Out[4]:
```
8x5 Array{Float64,2}:
 -3.55271e-15   1.77636e-15   8.88178e-16  -2.22045e-16  -2.22045e-16
 -7.10543e-15  -1.77636e-15   4.44089e-16  -6.66134e-16   1.34615e-15
 -8.88178e-16   3.55271e-15   4.44089e-16   0.0          -1.77636e-15
 -8.88178e-16  -8.88178e-16   1.77636e-15   0.0           3.55271e-15
  8.88178e-16  -1.77636e-15   0.0          -6.66134e-16  -4.16334e-16
  0.0          -3.55271e-15   8.88178e-16   4.44089e-16   4.44089e-16
 -1.77636e-15  -8.88178e-16   0.0          -8.88178e-16  -3.55271e-15
  4.44089e-16   8.88178e-16  -7.10543e-15  -1.77636e-15  -1.33227e-15
```

9.3.4 Example - Strongly scaled matrix

In [5]:
```
m=20
n=15
B=rand(m,n)
D=exp(50*(rand(n)-0.5))
As=B*diagm(D)
```

Out[5]:
```
20x15 Array{Float64,2}:
 2.10138e-8  0.00100182   2.79511e-8  …  5.75259e-11  3.04789e8  3.02756e10
 1.12477e-8  9.40967e-6   1.13065e-8     3.54726e-11  4.3182e8   3.4384e10
 4.68631e-9  0.000184595  2.83483e-8     4.1922e-11   7.85035e7  5.90449e10
 5.64317e-8  0.00119679   1.21429e-8     3.95774e-13  4.15415e8  6.24004e10
 8.57762e-9  0.000330184  2.70371e-8     1.76082e-11  2.03152e8  1.37779e10
 1.07354e-8  0.00118288   2.79863e-8  …  1.03369e-10  4.34209e8  4.23691e10
 3.42882e-8  5.39102e-5   1.63994e-8     7.63858e-11  4.78527e7  4.21632e10
 6.03348e-8  0.0011452    9.51067e-9     8.25521e-11  4.00684e8  5.2353e10
 2.89983e-8  8.22379e-5   2.36814e-8     7.92727e-11  2.60421e8  4.00863e10
 2.39828e-9  0.000358521  3.11818e-8     7.53429e-11  1.45195e8  1.93041e10
 1.71788e-9  5.77005e-5   3.03042e-8  …  9.27558e-11  2.15356e8  5.67653e10
 1.42274e-8  0.000159815  5.85584e-9     9.85951e-11  2.50143e8  1.17916e10
 1.50497e-8  0.000912344  2.86018e-8     8.16801e-12  1.75202e8  5.32017e10
 3.09749e-8  0.000959101  3.50519e-8     6.00976e-11  4.40889e8  3.30819e10
 5.84868e-8  0.000321227  2.06258e-8     9.60606e-11  2.7594e8   3.35634e10
 2.01261e-8  0.000568321  5.80715e-9  …  8.08718e-11  1.4182e8   6.19023e10
 1.25897e-8  0.00112786   3.02016e-8     5.22049e-11  2.92812e8  5.80208e10
 3.71633e-8  0.000331393  6.53319e-9     8.22921e-11  3.02046e8  3.58206e10
 2.41793e-8  0.000222238  2.8295e-8      7.1991e-12   2.65225e8  4.18035e10
 3.75646e-8  9.11609e-5   7.47371e-9     2.30195e-11  1.51586e8  3.42236e10
```

In [6]:
```
cond(B), cond(As)
```

Out[6]:
```
(42.22579886642498,1.4075147572949901e22)
```

In [7]:
```
Us,σs,Vs=myJacobiR(As)
```

```
Out[7]: (
        20x15 Array{Float64,2}:
         -0.054113    0.234        0.303614    …  -0.196129    0.202689   0.155776
          0.0841474  -0.568624    -0.154565      -0.245218    0.359919   0.176918
         -0.31744    -0.205659     0.149033       0.0977926  -0.385911   0.303787
          0.137277    0.0785212   -0.00735042     0.0561339   0.0955616  0.321061
         -0.014524    0.0214109    0.350976       0.147304    0.189829   0.0708931
         -0.174746    0.0743872   -0.00762322  …   0.198393    0.295271   0.218001
          0.341975    0.0247038    0.476789      -0.190291   -0.287999   0.21693
          0.481862    0.155584    -0.116422       0.184828    0.159153   0.269367
          0.0497351  -0.19969      0.301013       0.0898352   0.051632   0.206251
         -0.195401    0.287552    -0.299258       0.22397     0.0548243  0.0993234
         -0.374069   -0.326663     0.0236574   …   0.384756   -0.159196   0.292063
          0.122993   -0.0652242   -0.123933       0.237525    0.277964   0.0606751
          0.050188    0.380145    -0.135403      -0.0375881  -0.189534   0.273727
         -0.376592    0.251041     0.101865      -0.0839251   0.384783   0.170218
          0.202007   -0.0672985    0.198327       0.228678    0.130897   0.172691
          0.123439    0.059564    -0.359441    …   0.0996266  -0.314463   0.31849
         -0.153114    0.202802     0.0513148     -0.341893   -0.0526452  0.298525
          0.0896449  -0.0404625   -0.146045      -0.0533807   0.151131   0.184305
         -0.166196   -0.114806    -0.20023       -0.535706    0.0442267  0.215086
          0.187876   -0.193807    -0.2004        -0.131041   -0.0630473  0.176084 ,

        [6.405440634638585e-8,0.0013890235147308923,2.3111763393400786e-8,3.611563544824771
        15x15 Array{Float64,2}:
          0.855209      6.02449e-6    0.510493     …   5.41722e-17   5.52338e-19
         -2.8625e-6     0.999999     -1.10773e-5       1.42588e-12  1.22952e-14
         -0.511041      9.74099e-6    0.856009         2.87635e-17  4.47715e-19
          0.00695793   -4.61607e-7    0.0814193        1.90077e-18  3.29953e-20
         -0.08563      -6.0877e-5    -0.00345818       1.52454e-16  1.42946e-18
         -4.98251e-10   3.45396e-6   -3.82927e-10  …   3.79727e-8   5.7647e-10
         -0.000165692   1.75836e-9    0.000240225      5.08471e-21  1.79801e-22
         -4.17245e-10   0.000148532  -9.46922e-9       2.59089e-9   4.19434e-11
          0.00853831   -0.000295309   0.00172596       3.54736e-15  4.27872e-17
         -1.93123e-11  -4.30052e-7   -2.81367e-11      7.12483e-8   1.92497e-9
         -0.000260152  -3.0947e-8     0.000754375  …   7.99223e-20  1.18509e-21
          8.0724e-8     0.00103509    2.55964e-7       5.60927e-11  9.22682e-13
          0.000164355  -3.51655e-9    0.000922635      1.08161e-19  1.22844e-21
         -2.79464e-17  -1.96937e-12  -1.5511e-17       0.999984     0.00564579
         -2.63239e-20  -9.52493e-15  -7.71333e-20     -0.00564579   0.999984   )

In [8]: [sort(σs,rev=true) svdvals(As)]

Out[8]: 15x2 Array{Float64,2}:
           1.94361e11     1.94361e11
           6.60394e8      6.60394e8
         271.628        271.628
          58.2739        58.2739
           4.35489        4.35489
           0.0740062      0.0740062
           0.00138902     0.00138542
           5.3655e-6      5.36663e-6
```

```
 1.37558e-7     1.37675e-7
 6.40544e-8     6.77055e-8
 2.31118e-8     3.50172e-8
 3.61156e-9     3.74239e-9
 8.75354e-11    8.78194e-11
 4.60718e-11    4.85273e-11
 1.36641e-11    1.38088e-11
```

```
In [9]: As*Vs-Us*diagm(σs)
```

```
Out[9]: 20x15 Array{Float64,2}:
  2.89513e-24  -4.33681e-19  -1.32349e-23  …  -7.45058e-8   3.8147e-6
  8.27181e-25   0.0          -1.15805e-23     -2.98023e-8   0.0
  0.0           2.71051e-19  -4.1359e-25      -2.98023e-8   0.0
  5.62483e-23   8.13152e-20   3.36042e-25     -5.21541e-8   0.0
 -8.16841e-24  -2.71051e-20  -9.92617e-24     -1.49012e-8  -1.90735e-6
  1.65436e-24   3.93023e-19   2.50739e-24  …   0.0          0.0
  2.64698e-23  -2.71051e-19  -1.32349e-23      0.0          0.0
  5.29396e-23   8.13152e-20   4.1359e-24      -4.47035e-8   0.0
  3.01921e-23  -3.79471e-19   8.27181e-25     -4.47035e-8   0.0
  3.30872e-24  -1.6263e-19    1.07533e-23     -7.45058e-9   0.0
  0.0          -3.79471e-19   7.85822e-24  …  -4.47035e-8   0.0
  6.61744e-24   1.35525e-20  -7.44463e-24      0.0          0.0
  1.15805e-23   1.0842e-19    9.09899e-24     -1.19209e-7   0.0
  2.97785e-23  -5.96311e-19  -8.27181e-24      0.0          0.0
  5.62483e-23   6.77626e-20   3.30872e-24     -7.45058e-8   3.8147e-6
  1.32349e-23  -3.11708e-19  -1.98523e-23  …  -8.9407e-8    0.0
  4.96308e-24   5.42101e-20  -9.71937e-24     -1.04308e-7   7.62939e-6
  4.71493e-23  -3.45589e-19  -6.20385e-24     -5.96046e-8   0.0
  1.32349e-23  -2.43945e-19  -7.44463e-24      7.45058e-9   0.0
  3.63959e-23  -5.42101e-20  -1.40621e-23     -5.21541e-8   3.8147e-6
```

In the alternative approach, we first apply QR factorization with column pivoting to obtain the square matrix.

```
In [10]: Q,R,p=qr(As,Val{true},thin=true)
```

```
Out[10]: (
 20x15 Array{Float64,2}:
  -0.155772   -0.202692    0.20421    …   0.197001   -0.0656407   0.228358
  -0.176911   -0.359922    0.22284        0.239878    0.0713883   0.11542
  -0.303795    0.385905   -0.13301       -0.107947    0.147768    0.103079
  -0.32106    -0.0955677   0.243543      -0.0409449  -0.212693   -0.211565
  -0.0708895  -0.18983    -0.0575801     -0.157713    0.214074   -0.155135
  -0.217995   -0.295275    0.0155181  …  -0.169931   -0.466432   -0.139071
  -0.216936    0.287995   -0.107296       0.200262   -0.182674    0.062717
  -0.269364   -0.159158    0.0952053     -0.206215    0.268669    0.396004
  -0.20625    -0.0516359   0.0237242     -0.068307   -0.35145    -0.100947
  -0.0993223  -0.0548262  -0.182266      -0.215948   -0.110234   -0.145479
  -0.292066    0.159191    0.262777   …  -0.391743    0.0897899   0.0822886
  -0.0606697  -0.277965   -0.265459      -0.242998    0.0922273  -0.014071
  -0.273731    0.189529    0.332391       0.0409619  -0.0524031  -0.0211061
```

```
 -0.170211   -0.384786   -0.274466       0.0807805  -0.0239114   0.328506
 -0.172689   -0.1309     -0.384818      -0.242683    0.252384   -0.0677353
 -0.318496    0.314457   -0.139551   ...  -0.0994635  -0.0651598   0.238197
 -0.298526    0.0526394  -0.0984955      0.322415    0.451737   -0.417332
 -0.184302   -0.151134    0.245347       0.0482756   0.196081   -0.433264
 -0.215085   -0.0442309  -0.141065       0.532742    0.0153204   0.215961
 -0.176086    0.0630439  -0.437069       0.151802   -0.297211   -0.241767 ,

15x15 Array{Float64,2}:
 -1.94358e11  -1.09731e9  -374.137   -112.043   ...  -2.30334e-10  -3.49461e-11
  0.0         -6.60404e8   -47.0591   -25.0791     -5.27846e-11  -3.35858e-12
  0.0          0.0        -270.501    -24.0909     -5.03085e-11  -1.09282e-11
  0.0          0.0           0.0      -58.504      -4.02195e-11  -9.02763e-12
  0.0          0.0           0.0        0.0        -3.09713e-11  -1.24398e-12
  0.0          0.0           0.0        0.0     ...  -1.52301e-11  -7.88351e-13
  0.0          0.0           0.0        0.0        -4.29858e-11   2.44243e-12
  0.0          0.0           0.0        0.0         9.93415e-12   8.5305e-12
  0.0          0.0           0.0        0.0         4.55473e-11  -3.86396e-12
  0.0          0.0           0.0        0.0        -1.44813e-11  -9.09405e-12
  0.0          0.0           0.0        0.0     ...  -2.07688e-11  -7.53853e-12
  0.0          0.0           0.0        0.0         1.02875e-11   6.76388e-12
  0.0          0.0           0.0        0.0         7.14295e-12   7.79646e-12
  0.0          0.0           0.0        0.0         4.62406e-11  -1.5475e-12
  0.0          0.0           0.0        0.0         0.0           1.37328e-11,

[15,14,10,6,8,12,2,9,5,1,3,4,13,11,7])
```

```
In [11]: UR,σR,VR=myJacobiR(R')
```

```
Out[11]: (
15x15 Array{Float64,2}:
 -0.999984      0.00564579    1.54988e-9   ...   8.89265e-23   1.25632e-22
 -0.00564579   -0.999984      7.44936e-8      -3.28616e-20   1.09512e-20
 -1.92497e-9   -7.12483e-8   -0.995652        -2.25304e-13  -5.67278e-14
 -5.7647e-10   -3.79727e-8   -0.0929812       -7.26676e-13  -1.24669e-13
 -4.19434e-11  -2.59089e-9   -0.0055855        6.92141e-12   1.85812e-12
 -9.22682e-13  -5.60927e-11  -9.32094e-5   ...  -6.19205e-11  -1.25377e-10
 -1.22952e-14  -1.42588e-12   8.19077e-7       5.82373e-8    2.55026e-9
 -4.27872e-17  -3.54736e-15  -5.13362e-9      -5.91428e-6   -1.01214e-6
 -1.42946e-18  -1.52454e-16  -4.23981e-10      0.00039262    4.20606e-5
 -5.52338e-19  -5.41722e-17  -8.26115e-11     -4.06701e-5    7.72685e-5
 -4.47715e-19  -2.87635e-17  -3.94827e-11  ...  -0.000634816  -0.000222733
 -3.29953e-20  -1.90077e-18  -2.62211e-12     -0.00333725   -0.00249507
 -1.22844e-21  -1.08161e-19  -2.12113e-13      0.107886      0.0947872
 -1.18509e-21  -7.99223e-20  -1.88146e-13      0.993025      0.0372247
 -1.79801e-22  -5.08471e-21  -4.08894e-14     -0.0474466     0.994798    ,

[1.9436095702753955e11,6.603936993123478e8,271.6282618948315,58.27389433176226,4.35
15x15 Array{Float64,2}:
  1.0          -1.91834e-5   -2.16606e-18  ...  -2.10797e-44  -8.83243e-45
  1.91834e-5    1.0          -3.06361e-14      2.29256e-39  -2.2657e-40
  2.75085e-18   3.0575e-14    0.9998           3.83738e-26   2.86555e-27
```

```
 1.73276e-19   3.35953e-15   0.0200032        5.56455e-25   2.79375e-26
 9.40178e-22   1.70923e-17   8.95808e-5      -7.30817e-23  -5.823e-24
-3.51347e-25  -6.28624e-21  -2.53973e-8   ...  -3.85967e-20  -2.31514e-20
-8.78691e-29  -2.9991e-24    4.18852e-12      1.93164e-15   2.50861e-17
 0.0          -2.88326e-29  -1.01502e-16     -5.07905e-11  -2.57792e-12
 0.0           0.0           2.15964e-19     -1.31895e-7   -4.19115e-9
 0.0           0.0          -1.30828e-20     -5.23105e-8    1.81114e-8
 0.0           0.0           3.83022e-21  ...   1.14888e-6    9.59303e-8
 0.0           0.0          -3.5001e-23      -4.18855e-5   -9.36847e-6
 0.0           0.0           6.16459e-26     -0.0572863    -0.0149269
 0.0           0.0          -3.1796e-26       0.998258      0.0133078
 0.0           0.0          -2.06726e-27     -0.0141426     0.9998      )
```

```
In [12]: (sort(σs)-sort(σR))./sort(σs)
```

```
Out[12]: 15-element Array{Float64,1}:
          1.3006e-15
          1.82347e-15
          5.90604e-16
          3.43555e-16
          1.43162e-16
          6.19859e-16
          9.62134e-16
          1.8944e-15
          2.65387e-15
          5.62566e-16
          1.2237e-15
          9.75453e-16
          4.18538e-16
          0.0
          1.57015e-16
```

```
In [13]: P=eye(15)
         P=P[:,p];
```

Now $QRP^T = A$ and $R^T = U_R \Sigma_R V_R^T$, so $A = (QV_R)\Sigma_R(U_R^T P^T)$ is an SVD of A.

```
In [14]: # Check the residual
         U1=Q*VR
         V1=UR[invperm(p),:]
         norm(As*V1-U1*diagm(σR))
```

```
Out[14]: 2.9134940798592712e-5
```

9.4 Lanczos method

The function `svds()` is based on the Lanczos method for symmetric matrices. Input can be matrix, but also an operator which defines the product of the given matrix with a vector.

```
In [15]: ?svds
```

```
search: svds svdvals svdvals! is_valid_ascii svd svdfact svdfact! isvalid
```

Out[15]:

```
..  svds(A; nsv=6, ritzvec=true, tol=0.0, maxiter=1000) -> (left_sv, s, right_sv, nconv, ni

``svds`` computes largest singular values ``s`` of ``A`` using Lanczos or Arnoldi iteration
Uses :func:`eigs` underneath.

Inputs are:

* ``A``: Linear operator. It can either subtype of ``AbstractArray`` (e.g., sparse matrix) o
* ``nsv``: Number of singular values.
* ``ritzvec``: Whether to return the left and right singular vectors ``left_sv`` and ``right
* ``tol``: tolerance, see :func:`eigs`.
* ``maxiter``: Maximum number of iterations, see :func:`eigs`.

**Example**::

   X = sprand(10, 5, 0.2)
   svds(X, nsv = 2)
```

In [16]:

```
m=20
n=15
A=rand(m,n);
```

In [17]:

```
U,σ,V=svd(A)
```

Out[17]:

```
(
20x15 Array{Float64,2}:
 -0.240339  -0.00591555   0.140354    …  -0.301328    -0.00365383
 -0.204612  -0.11566      0.465815       -0.228446    -0.19806
 -0.23935    0.0914358    0.00370841      0.256475    -0.274818
 -0.272593   0.188513     0.0379744      -0.275999     0.00880543
 -0.180147  -0.43128     -0.365829       -0.166577     0.0386427
 -0.199518  -0.241917     0.192626    …  -0.166434    -0.0888854
 -0.273135   0.00845405  -0.244867        0.403096    -0.0299702
 -0.194973  -0.0768215    0.20291        -0.19143      0.352643
 -0.248844  -0.00642136  -0.356442       -0.328271    -0.151218
 -0.241481   0.464142    -0.114384       -0.245256    -0.053765
 -0.184648   0.0555143    0.0677048   …   0.204391    -0.27974
 -0.164448  -0.279704    -0.124711        0.00319137   0.294488
 -0.212239  -0.310288    -0.139027        0.1214      -0.302567
 -0.273388  -0.235969     0.298433        0.316283     0.431367
 -0.181014   0.274129     0.142019        0.271869    -0.143367
 -0.178598   0.329724    -0.338658    …  -0.0134783    0.437011
 -0.214514  -0.0826803   -0.0985989       0.0473036   -0.0965249
 -0.280683  -0.0157228   -0.112096        0.190112     0.0190352
 -0.200248   0.0380008    0.105538       -0.107869    -0.141319
 -0.230598   0.225951     0.229882        0.109073     0.202397  ,
```

```
[9.233351437504982,2.004620481347783,1.779692715123704,1.7127885402637306,1.537210
15x15 Array{Float64,2}:
 -0.313254  -0.167464    0.307605   -0.0281782   …  -0.0810468    -0.258126
 -0.215729   0.296844   -0.418633    0.175621        0.299936      0.144254
 -0.330073   0.0917875   0.313287   -0.161366       -0.519163      0.374629
 -0.205718  -0.480693    0.182506    0.416094        0.167803      0.0199484
 -0.201695   0.0203653   0.0286203  -0.478183        0.227305     -0.168919
 -0.252696  -0.173372   -0.459606   -0.356151    …  -0.404493     -0.0125347
 -0.2403     0.0670156  -0.356396    0.0456847       0.0725507     0.274877
 -0.286046   0.209965    0.403473    0.00381201      0.247531      0.194413
 -0.206989   0.105898    0.0219257   0.386024       -0.210908     -0.162787
 -0.25608   -0.665213   -0.119927   -0.0854967       0.15367       0.204668
 -0.279379   0.209269   -0.0380944   0.344915    …  -0.290018      0.0767328
 -0.202823   0.136425   -0.0567194  -0.0409475       0.247537      0.363866
 -0.281256   0.171238    0.0148693  -0.211457       -0.00119156   -0.4499
 -0.268583  -0.0189101  -0.23668     0.258805        0.0250078    -0.460736
 -0.284797   0.139612    0.145029   -0.151411        0.331444     -0.0917052)
```

In [18]:
```
# All singular values
UL,σL,VL, rest=svds(A,nsv=15);
```

In [19]:
```
(σ-σL)./σ
```

Out[19]:
```
15-element Array{Float64,1}:
 -3.8477e-16
 -8.86131e-16
 -1.24766e-15
  1.03711e-15
 -1.15557e-15
  8.19575e-16
 -1.2321e-15
 -8.0228e-16
 -3.50799e-16
  0.0
 -1.63141e-16
  0.0
  2.07754e-16
  0.0
 -1.07329e-15
```

In [20]:
```
# Some largest singular values
Up,σp,Vp, rest=svds(A,nsv=5);
(σ[1:5]-σp)./σ[1:5]
```

Out[20]:
```
5-element Array{Float64,1}:
  1.92385e-16
 -1.55073e-15
 -2.62008e-15
  2.59278e-16
  5.77786e-16
```

9.4.1 Example - Large matrix

```
In [21]: m=2000
         n=1500
         Ab=rand(m,n);

In [22]: @time Ub,σb,Vb=svd(Ab);

  3.580198 seconds (185 allocations: 131.801 MB, 0.45% gc time)


In [23]: # This is rather slow
         @time Ul,σl,rest=svds(Ab,nsv=10);

  2.469470 seconds (9.37 k allocations: 60.580 MB, 0.14% gc time)


In [24]: (σb[1:10]-σl)./σb[1:10]

Out[24]: 10-element Array{Float64,1}:
          -5.77484e-15
          -5.00912e-15
           3.41555e-15
           2.98601e-15
          -8.98982e-16
          -1.04944e-14
           1.80086e-15
          -2.55931e-15
           1.35796e-15
           1.0584e-15
```

9.4.2 Example - Very large sparse matrix

```
In [25]: ?sprand

search: sprand sprandn sprandbool StepRange



Out[25]:

..  sprand([rng,] m,n,p [,rfn])

Create a random ``m`` by ``n`` sparse matrix, in which the probability of any
element being nonzero is independently given by ``p`` (and hence the mean
density of nonzeros is also exactly ``p``). Nonzero values are sampled from
the distribution specified by ``rfn``. The uniform distribution is used in
case ``rfn`` is not specified. The optional ``rng`` argument specifies a
random number generator, see :ref:`Random Numbers <random-numbers>`.

In [26]: m=10000
         n=3000
         A=sprand(m,n,0.05)
```

```
Out[26]: 10000x3000 sparse matrix with 1498624 Float64 entries:
               [37    ,     1]  =  0.214525
               [41    ,     1]  =  0.0924789
               [66    ,     1]  =  0.199703
               [106   ,     1]  =  0.794311
               [107   ,     1]  =  0.0675695
               [110   ,     1]  =  0.749529
               [113   ,     1]  =  0.793374
               [165   ,     1]  =  0.0574841
               [169   ,     1]  =  0.805687
               [187   ,     1]  =  0.415969
               ⋮
               [9781 ,  3000]  =  0.70868
               [9806 ,  3000]  =  0.618624
               [9894 ,  3000]  =  0.274889
               [9897 ,  3000]  =  0.473151
               [9898 ,  3000]  =  0.44278
               [9909 ,  3000]  =  0.121238
               [9927 ,  3000]  =  0.579126
               [9938 ,  3000]  =  0.397157
               [9958 ,  3000]  =  0.591276
               [9966 ,  3000]  =  0.173047
               [10000,  3000]  =  0.914066

In [27]: # No vectors, this takess about 30 sec.
         @time σ1,rest=svds(A,nsv=100,ritzvec=false)

 28.767826 seconds (239.26 k allocations: 670.163 MB, 0.18% gc time)


Out[27]: ([137.464,19.645,19.5688,19.5546,19.5177,19.5031,19.4921,19.4636,19.452,19.4352  .

In [28]: @time σ2=svdvals(full(A));

 20.988790 seconds (6.88 k allocations: 459.827 MB, 0.02% gc time)


In [29]: (σ1-σ2[1:100])./σ2[1:100]

Out[29]: 100-element Array{Float64,1}:
          -8.47703e-15
           5.06367e-15
          -4.17564e-15
          -1.27177e-14
          -7.82707e-15
           1.27513e-15
           1.09359e-15
          -3.83316e-15
          -3.6528e-15
          -3.10756e-15
          -1.64754e-15
```

```
-2.01511e-15
-1.4664e-15
 ⋮
-2.29821e-15
-2.10875e-15
-1.91749e-15
-2.68475e-15
-9.59983e-16
-3.07421e-15
-1.92222e-15
-1.92281e-15
-1.34617e-15
-2.11629e-15
-1.92483e-15
 1.92554e-16
```

```
In [ ]:
```

10 Algorithms for Structured Matrices

For matrices with some special structure, it is possible to derive versions of algorithms which are faster and/or more accurate than the standard algorithms.

10.1 Prerequisites

The reader should be familiar with concepts of eigenvalues and eigen vectors, singular values and singular vectors, related perturbation theory, and algorithms.

10.2 Competences

The reader should be able to recognise matrices which have rank-revealing decomposition and apply adequate algorithms, and to apply forward stable algorithms to arrowhead and diagonal-plus-rank-one matrices.

10.3 Rank revealing decompositions

For more details, see Z. Drmač, Computing Eigenvalues and Singular Values to High Relative Accuracy and J. Demmel et al, Computing the singular value decomposition with high relative accuracy and the references therein.

Let $A \in \mathbb{R}^{m \times n}$ with $\operatorname{rank}(A) = n$ (therefore, $m \geq n$) and $A = U\Sigma V^T$ its thin SVD.

10.3.1 Definitions

Let $A \in \mathbb{R}^{m \times n}$.

The singular values of A are (**perfectly**) **well determined to high relative accuracy** if changing any entry A_{kl} to θA_{kl}, $\theta \neq 0$, causes perturbations in singular values bounded by $\min\{|\theta|, 1/|\theta|\}\sigma_j \leq \tilde{\sigma}_j \leq \max\{|\theta|, 1/|\theta|\}\sigma_j$ for all j.

The **sparsity pattern** of A, $Struct(A)$, is the set of indices for which A_{kl} is permitted to be non-zero.

The **bipartite graph** of the sparsity pattern S, $\mathcal{G}(S)$, is the graph with vertices partitioned into row vertices $r_1, \ldots, r_m$ and column vertices $c_1, \ldots, c_n$, where r_k and c_l are connected if and only if $(k, l) \in S$.

If $\mathcal{G}(S)$ is acyclic, matrices with sparsity pattern S are **biacyclic**.

A decomposition $A = XDY^T$ with diagonal matrix D is called a **rank revealing decomposition** (RRD) if X and Y are full-column rank well-conditioned matrices.

Hilbert matrix is a square matrix H with elements $H_{ij} = \dfrac{1}{i+j-1}$.

Hankel matrix is a square matrix with constant elements along skew-diagonals.

Cauchy matrix is an $m \times n$ matrix C with elements $C_{ij} = \dfrac{1}{x_i + y_j}$ with $x_i + y_j \neq 0$ for all i, j.

10.3.2 Facts

1. The singular values of A are perfectly well determined to high relative accuracy if and only if the bipartite graph $\mathcal{G}(S)$ is acyclic (forest of trees). Examples are bidiagonal and

arrowhead matrices. Sparsity pattern S of acyclic bipartite graph allows at most $m+n-1$ nonzero entries. A bisection algorithm computes all singular values of biacyclic matrices to high relative accuracy.

2. An RRD of A can be given or computed to high accuracy by some method. Typical methods are Gaussian elimination with complete pivoting or QR factorization with complete pivoting.

3. Let $\hat{X}\hat{D}\hat{Y}^T$ be the computed RRD of A satisfying $|D_{jj}-\hat{D}_{jj}|\leq O(\varepsilon)|D_{jj}|$, $\|X-\hat{X}\|\leq O(\varepsilon)\|X\|$, and $\|Y-\hat{Y}\|\leq O(\varepsilon)\|Y\|$.

    1. Perform QR factorization with pivoting to get $\hat{X}\hat{D}=QRP$, where P is a permutation matrix. Thus $A=QRP\hat{Y}^T$.
    2. Multiply $W=RP\hat{Y}^T$ (*NOT* Strassen's multiplication). Thus $A=QW$ and W is well-scaled from the left.
    3. Compute the SVD of $W^T=V\Sigma^T\bar{U}^T$ using one-sided Jacobi method. Thus $A=Q\bar{U}\Sigma V^T$.
    4. Multiply $U=Q\bar{U}$. Thus $A=U\Sigma V^T$ is the computed SVD of A.

4. Let $R=D'R'$, where D' is such that the *rows* of R' have unit norms. Then the following error bounds hold:

$$\frac{|\sigma_j-\tilde{\sigma}_j|}{\sigma_j}\leq O(\varepsilon\kappa(R')\cdot\max\{\kappa(X),\kappa(Y)\})\leq O(\varepsilon n^{3/2}\kappa(X)\cdot\max\{\kappa(X),\kappa(Y)\}).$$

5. Hilbert matrix is Hankel matrix and Cauchy matrix, it is symmetrixc positive definite and *very* ill-conditioned.

6. Every sumbatrix of a Cauchy matrix is itself a Cauchy matrix.

7. Determinat of a square Cauchy matrix is

$$\det(C)=\frac{\prod_{1\leq i<j\leq n}(x_j-x_i)(y_j-y_i)}{\prod_{1\leq i,j\leq n}(x_i+y_j)}.$$

    It is computed with elementwise high relative accuracy.

8. Let A be square and nonsingular and let $A=LDR$ be its decomposition with diagonal D, lower unit-triangular L, and upper unit-triangular R. The closed formulas using quotients of minors are (see A. S. Householder, The Theory of Matrices in Numerical Analysis):

$$\begin{aligned}
D_{11}&=A_{11},\\
D_{jj}&=\frac{\det(A_{1:j,1:j})}{\det(A_{1:j-1,1:j-1})},\quad j=2,\ldots,n,\\
L_{jj}&=1,\\
L_{ij}&=\frac{\det(A_{[1,2,\ldots,j-1,i],[1:j]})}{\det(A_{1:j,1:j})},\quad j<i,\\
R_{jj}&=1,\\
R_{ji}&=\frac{\det(A_{[1,2,\ldots,j],[1,2,\ldots,j-1,i]})}{\det(A_{1:j,1:j})},\quad i>j,
\end{aligned}$$

10.3.3 Example - Positive definite matrix

Let $A = DA_S D$ be strongly scaled symmetric positive definite matrix. Then Cholesky factorization with complete (diagonal) pivoting is an RRD. Consider the following three step algorithm:

1. Compute $P^T AP = LL^T$ (*Cholesky factorization with complete pivoting*).
2. Compute the $L = \bar{U}\Sigma V^T$ (*one-sided Jacobi, V is not needed*).
3. Set $\Lambda = \Sigma^2$ and $U = P\bar{U}$. Thus $A = U\Lambda U^T$ is an EVD of A.

The Cholesky factorization with pivoting can be implemented very fast with block algorithm (see C. Lucas, LAPack-Style Codes for Level 2 and 3 Pivoted Cholesky Factorizations).
The eigenvalues $\tilde{\lambda}_j$ computed using the above algorithm satisfy relative error bounds:

$$\frac{|\lambda_j - \tilde{\lambda}_j|}{\lambda_j} \leq O(n\varepsilon \|A_S\|_2^{-1}).$$

```
In [1]: include("ModuleB.jl")
        using ModuleB
```

```
In [2]: n=20
        B=randn(n,n)
        # Scaled matrix
        As=full(Symmetric(B'*B))
        # Scaling
        D=exp(50*(rand(n)-0.5))
        # Parentheses are necessary!
        A=map(Float64,[As[i,j]*(D[i]*D[j]) for i=1:n, j=1:n])
        issym(A), cond(As), cond(A)
```

```
Out[2]: (true,2583.8909256820393,6.98609524271529e30)
```

```
In [3]: ?chol
```

```
search: chol cholfact cholfact! searchsortedlast chop chomp chmod Cshort
```

```
Out[3]:
```

```
chol(A, [LU]) -> F
```

Compute the Cholesky factorization of a symmetric positive definite matrix `A` and return the matrix `F`. If `LU` is `Val:U` (Upper), `F` is of type `UpperTriangular` and `A = F'*F`. If `LU` is `Val:L` (Lower), `F` is of type `LowerTriangular` and `A = F*F'`. `LU` defaults to `Val:U`.
We will not use the Cholesky factorization with complete pivoting. Instead, we will just sort the diagonal of A in advance, which is sufficient for this example.
Write the function for Cholesky factorization with complete pivoting as an excercise.

```
In [4]: ?sortperm
```

```
search: sortperm sortperm!
```

Out[4]:

```
..  sortperm(v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])

Return a permutation vector of indices of ``v`` that puts it in sorted order.
Specify ``alg`` to choose a particular sorting algorithm (see Sorting Algorithms).
``MergeSort`` is used by default, and since it is stable, the resulting permutation
will be the lexicographically first one that puts the input array into sorted order {
i.e. indices of equal elements appear in ascending order. If you choose a non-stable
sorting algorithm such as ``QuickSort``, a different permutation that puts the array
into order may be returned. The order is specified using the same keywords as ``sort!``.

See also :func:`sortperm!`
```

In [5]:
```
p=sortperm(diag(A), rev=true)
L=chol(A[p,p])
```

Out[5]:
```
20x20 UpperTriangular{Float64,Array{Float64,2}}:
 1.43558e9  2.4712e7   5.31533e6   3.01695e6  …   0.000100814   2.23552e-7
 0.0        1.77309e8  2.92828e7  -1.87192e6      3.94249e-5    1.56376e-7
 0.0        0.0        6.48179e7   7.86315e6     -5.89792e-5    1.91929e-7
 0.0        0.0        0.0         3.03605e7     -1.68491e-5    2.20755e-7
 0.0        0.0        0.0         0.0           -8.11625e-5   -3.63825e-7
 0.0        0.0        0.0         0.0        …  -7.43661e-5    7.95642e-8
 0.0        0.0        0.0         0.0           -2.68143e-5   -3.59289e-7
 0.0        0.0        0.0         0.0            5.26835e-5    4.11275e-8
 0.0        0.0        0.0         0.0            9.92938e-7    3.03166e-7
 0.0        0.0        0.0         0.0           -3.74084e-6   -2.22465e-7
 0.0        0.0        0.0         0.0        …   0.00011929    7.02092e-7
 0.0        0.0        0.0         0.0            4.62455e-5   -4.487e-7
 0.0        0.0        0.0         0.0           -2.32847e-5    1.61081e-8
 0.0        0.0        0.0         0.0           -2.36166e-5    4.34081e-7
 0.0        0.0        0.0         0.0            3.52997e-5   -1.98937e-7
 0.0        0.0        0.0         0.0        …   2.86242e-5   -6.86283e-7
 0.0        0.0        0.0         0.0            1.32551e-5    3.19534e-7
 0.0        0.0        0.0         0.0           -9.55869e-5   -4.27361e-7
 0.0        0.0        0.0         0.0            3.41928e-5   -5.83852e-8
 0.0        0.0        0.0         0.0            0.0           6.79193e-7
```

In [6]:
```
U,σ,V=myJacobiR(full(L'));
```

In [7]:
```
λ=σ.^2
U1=U[invperm(p),:]
λ, U1'*A*U1
```

Out[7]:
```
([2.061560582882486e18,3.2438331534735748e16,4.187811544997185e15,1.071876580779014e
20x20 Array{Float64,2}:
    2.06156e18    10.2227       0.0078125   …   1.35856        321.605
   10.2282         3.24383e16   0.095459        8.43683         30.5406
    0.943161      -0.0250397    4.18781e15     -2.59875e-6      -0.0843901
    1.51929       -0.0812988    0.0722656      -2.1415e-8       -0.0879212
 -251.423         -0.278793     0.0785522      -8.38497e-10     -0.0104345
```

```
-466.162         -0.0201416   -0.0487061    ...   6.82121e-13    0.00263036
 175.953          0.0738716    0.00842285        -0.00402913    -1.75189e-5
  -6.6838        -0.00230583  -0.00230249        -1.13569e-13    4.2439e-5
   2.9371         0.00413306   0.000967331        4.54481e-6    -3.56661e-8
  -0.045196      -0.00130051  -2.11208e-5         4.35205e-9     1.60284e-9
  -0.0150063    -11.4939      -9.80918e-5   ...   3.72026e-15    1.76841e-8
  -0.0106753     -6.9873      -9.26259e-7        -3.43777e-12    1.07209e-13
   0.00387572     2.7122       3.53972e-6         1.45942e-12    1.71317e-16
   0.00155322     1.00417     -0.439478           2.72651e-16    9.54724e-16
   0.00251477     1.78265      3.52184            4.64223e-16    1.60647e-15
   0.000368915    0.294624     1.94277      ...   7.31618e-17    2.37654e-16
   6.75658e-6     0.00427136  -0.00412643         1.10169e-18    4.1046e-18
   0.770967      -2.27035     -1.75293e-6        -5.90407e-16   -2.01726e-15
   1.35856        8.43683     -2.59875e-6         5.72372e-10    8.1552e-15
 321.605         30.5406      -0.0843901          8.1552e-15     5.40218e-13)
```

10.3.4 Example - Hilbert matrix

We need the newest version of the package SpecialMatrices.jl.

```
In [8]: # Pkg.checkout("SpecialMatrices")
        using SpecialMatrices
```

```
In [9]: whos(SpecialMatrices)
```

```
                  Cauchy    180 bytes  DataType
               Circulant    168 bytes  DataType
               Companion    168 bytes  DataType
               Frobenius    180 bytes  DataType
                  Hankel    168 bytes  DataType
                 Hilbert    180 bytes  DataType
                   Kahan    244 bytes  DataType
                 Riemann    168 bytes  DataType
         SpecialMatrices   3457 bytes  Module
                  Strang    168 bytes  DataType
                Toeplitz    168 bytes  DataType
             Vandermonde    168 bytes  DataType
```

```
In [10]: C=Cauchy([1,2,3,4,5],[0,1,2,3,4])
```

```
Out[10]: 5x5 SpecialMatrices.Cauchy{Int64}:
          1.0       0.5       0.333333  0.25      0.2
          0.5       0.333333  0.25      0.2       0.166667
          0.333333  0.25      0.2       0.166667  0.142857
          0.25      0.2       0.166667  0.142857  0.125
          0.2       0.166667  0.142857  0.125     0.111111
```

```
In [11]: H=Hilbert(5)
```

```
Out[11]: SpecialMatrices.Hilbert{Rational{Int64}}(5,5)
```

```
In [12]: Hf=full(H)

Out[12]: 5x5 Array{Rational{Int64},2}:
          1//1  1//2  1//3  1//4  1//5
          1//2  1//3  1//4  1//5  1//6
          1//3  1//4  1//5  1//6  1//7
          1//4  1//5  1//6  1//7  1//8
          1//5  1//6  1//7  1//8  1//9

In [13]: # This is exact
         det(Hf)

Out[13]: 1//266716800000

In [14]: # Exact formula for the determinant of a Cauchy matrix from Fact 7.
         import Base.det
         function det{T}(C::Cauchy{T})
             n=length(C.x)
             F=triu([(C.x[j]-C.x[i])*(C.y[j]-C.y[i]) for i=1:n, j=1:n],1)
             num=prod(F[find(F)])
             den=prod([(C.x[i]+C.y[j]) for i=1:n, j=1:n])
             if isinteger(C.x)&isinteger(C.y)
                 return num//den
             else
                 return num/den
             end
         end

Out[14]: det (generic function with 19 methods)

In [15]: det(C)

Out[15]: 1//266716800000
```

We now compute componentwise highly accurate $A = LDL^T$ factorization of a Hilbert (Cauchy) matrix. Using `Rational` numbers gives high accuracy.

```
In [16]: # Exact LDLT factorization from Fact 8, no pivoting.
         function myLDLT{T}(C::Cauchy{T})
             n=length(C.x)
             D=Array(Rational{T},n)
             L=eye(Rational{T},n)
             δ=[det(Cauchy(C.x[1:j],C.y[1:j])) for j=1:n]
             D[1]=map(Rational{T},C[1,1])
             D[2:n]=δ[2:n]./δ[1:n-1]
             for i=2:n
                 for j=1:i-1
                     L[i,j]=det(Cauchy( C.x[[1:j-1;i]], C.y[1:j])) / δ[j]
                 end
             end
             L,D
         end
```

```
Out[16]: myLDLT (generic function with 1 method)

In [17]: L,D=myLDLT(C)

Out[17]: (
         5x5 Array{Rational{Int64},2}:
          1//1  0//1    0//1  0//1  0//1
          1//2  1//1    0//1  0//1  0//1
          1//3  1//1    1//1  0//1  0//1
          1//4  9//10   3//2  1//1  0//1
          1//5  4//5   12//7  2//1  1//1,

         Rational{Int64}[1//1,1//12,1//180,1//2800,1//44100])

In [18]: L*diagm(D)*L' # -full(H)

Out[18]: 5x5 Array{Rational{Int64},2}:
          1//1  1//2  1//3  1//4  1//5
          1//2  1//3  1//4  1//5  1//6
          1//3  1//4  1//5  1//6  1//7
          1//4  1//5  1//6  1//7  1//8
          1//5  1//6  1//7  1//8  1//9

In [19]: # L*D*L' is an RRD
         cond(L)

Out[19]: 11.858249f0
```

We now compute the accurate EVD of the Hilbert matrix of order $n = 100$. We cannot use the function `myLDLT()` since the *computation of determinant causes overflow* and *there is no pivoting*. Instead, we use Algorithm 3 from J. Demmel, Computing the singular value decomposition with high relative accuracy.

```
In [20]: function myGECP(C::Cauchy)
             n=length(C.x)
             G=full(C)
             x=copy(C.x)
             y=copy(C.y)
             pr=collect(1:n)
             pc=collect(1:n)
             # Find the maximal element
             for k=1:n-1
                 i,j=ind2sub(size(G[k:n,k:n]),indmax(abs(G[k:n,k:n])))
                 i+=k-1
                 j+=k-1
                 if i!=k || j!=k
                     G[[i,k],:]=G[[k,i],:]
                     G[:, [j,k]]=G[:, [k,j]]
                     x[[k,i]]=x[[i,k]]
                     y[[k,j]]=y[[j,k]]
                     pr[[i,k]]=pr[[k,i]]
                     pc[[j,k]]=pc[[k,j]]
```

```
            end
            for r=k+1:n
                for s=k+1:n
                    G[r,s]=G[r,s]*(x[r]-x[k])*(y[s]-y[k])/
                    ((x[k]+y[s])*(x[r]+y[k]))
                end
            end
            G=full(Symmetric(G))
        end
        D=diag(G)
        X=tril(G,-1)*diagm(1.0./D)+I
        Y=diagm(1.0./D)*triu(G,1)+I
        X,D,Y', pr,pc
    end
```

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

In [21]:
```
# First a smaller test
l=8
C=Cauchy(collect(1:l),collect(0:l-1))
```

Out[21]:
```
8x8 SpecialMatrices.Cauchy{Int64}:
 1.0       0.5       0.333333  0.25       ...  0.166667   0.142857   0.125
 0.5       0.333333  0.25      0.2            0.142857   0.125      0.111111
 0.333333  0.25      0.2       0.166667       0.125      0.111111   0.1
 0.25      0.2       0.166667  0.142857       0.111111   0.1        0.0909091
 0.2       0.166667  0.142857  0.125          0.1        0.0909091  0.0833333
 0.166667  0.142857  0.125     0.111111   ...  0.0909091  0.0833333  0.0769231
 0.142857  0.125     0.111111  0.1            0.0833333  0.0769231  0.0714286
 0.125     0.111111  0.1       0.0909091      0.0769231  0.0714286  0.0666667
```

In [22]:
```
X,D,Y,pr,pc=myGECP(C)
```

Out[22]:
```
(
8x8 Array{Float64,2}:
 1.0       0.0        0.0       0.0        0.0        0.0        0.0       0.0
 0.333333  1.0        0.0       0.0        0.0        0.0        0.0       0.0
 0.125     0.65625    1.0       0.0        0.0        0.0        0.0       0.0
 0.5       0.9375    -0.47619   1.0        0.0        0.0        0.0       0.0
 0.2       0.857143   0.653061 -0.342857   1.0        0.0        0.0       0.0
 0.25      0.9375     0.38961  -0.327273   0.715909   1.0        0.0       0.0
 0.142857  0.714286   0.932945 -0.122449   0.487013  -0.952381   1.0       0.0
 0.166667  0.78125    0.824176 -0.247253   0.865385  -0.940171   0.712963  1.0,

[1.0,0.08888888888888889,0.012760416666666665,0.0023148148148148147,9.070294784580
8x8 Array{Float64,2}:
 1.0       0.0        0.0       0.0        0.0        0.0        0.0       0.0
 0.333333  1.0        0.0       0.0        0.0        0.0        0.0       0.0
 0.125     0.65625    1.0       0.0        0.0        0.0        0.0       0.0
 0.5       0.9375    -0.47619   1.0        0.0        0.0        0.0       0.0
 0.2       0.857143   0.653061 -0.342857   1.0        0.0        0.0       0.0
 0.25      0.9375     0.38961  -0.327273   0.715909   1.0        0.0       0.0
```

```
 0.142857  0.714286   0.932945  -0.122449  0.487013  -0.952381  1.0       0.0
 0.166667  0.78125    0.824176  -0.247253  0.865385  -0.940171  0.712963  1.0,

[1,3,8,2,5,4,7,6],[1,3,8,2,5,4,7,6])
```

```
In [23]: norm((X*diagm(D)*Y')-full(C)[pr,pc])

Out[23]: 8.488925965415388e-17

In [24]: norm(X[invperm(pr),:]*diagm(D)*Y[invperm(pc),:]'-full(C))

Out[24]: 8.488925965415385e-17

In [25]: # Now the big test.
         n=100
         H=Hilbert(n)
         C=Cauchy(collect(1:n), collect(0:n-1))

Out[25]: 100x100 SpecialMatrices.Cauchy{Int64}:
 1.0        0.5         0.333333    …  0.0102041   0.010101    0.01
 0.5        0.333333    0.25           0.010101    0.01        0.00990099
 0.333333   0.25        0.2            0.01        0.00990099  0.00980392
 0.25       0.2         0.166667       0.00990099  0.00980392  0.00970874
 0.2        0.166667    0.142857       0.00980392  0.00970874  0.00961538
 0.166667   0.142857    0.125       …  0.00970874  0.00961538  0.00952381
 0.142857   0.125       0.111111       0.00961538  0.00952381  0.00943396
 0.125      0.111111    0.1            0.00952381  0.00943396  0.00934579
 0.111111   0.1         0.0909091      0.00943396  0.00934579  0.00925926
 0.1        0.0909091   0.0833333      0.00934579  0.00925926  0.00917431
 0.0909091  0.0833333   0.0769231   …  0.00925926  0.00917431  0.00909091
 0.0833333  0.0769231   0.0714286      0.00917431  0.00909091  0.00900901
 0.0769231  0.0714286   0.0666667      0.00909091  0.00900901  0.00892857
 ⋮                                  ⋱
 0.011236   0.0111111   0.010989       0.00537634  0.00534759  0.00531915
 0.0111111  0.010989    0.0108696      0.00534759  0.00531915  0.00529101
 0.010989   0.0108696   0.0107527   …  0.00531915  0.00529101  0.00526316
 0.0108696  0.0107527   0.0106383      0.00529101  0.00526316  0.0052356
 0.0107527  0.0106383   0.0105263      0.00526316  0.0052356   0.00520833
 0.0106383  0.0105263   0.0104167      0.0052356   0.00520833  0.00518135
 0.0105263  0.0104167   0.0103093      0.00520833  0.00518135  0.00515464
 0.0104167  0.0103093   0.0102041   …  0.00518135  0.00515464  0.00512821
 0.0103093  0.0102041   0.010101       0.00515464  0.00512821  0.00510204
 0.0102041  0.010101    0.01           0.00512821  0.00510204  0.00507614
 0.010101   0.01        0.00990099     0.00510204  0.00507614  0.00505051
 0.01       0.00990099  0.00980392     0.00507614  0.00505051  0.00502513
```

We need a function to compute RRD from `myGECP()`

```
In [26]: function myRRD(C::Cauchy)
             X,D,Y,pr,pc=myGECP(C)
             X[invperm(pr),:], D, Y[invperm(pc),:]
         end
```

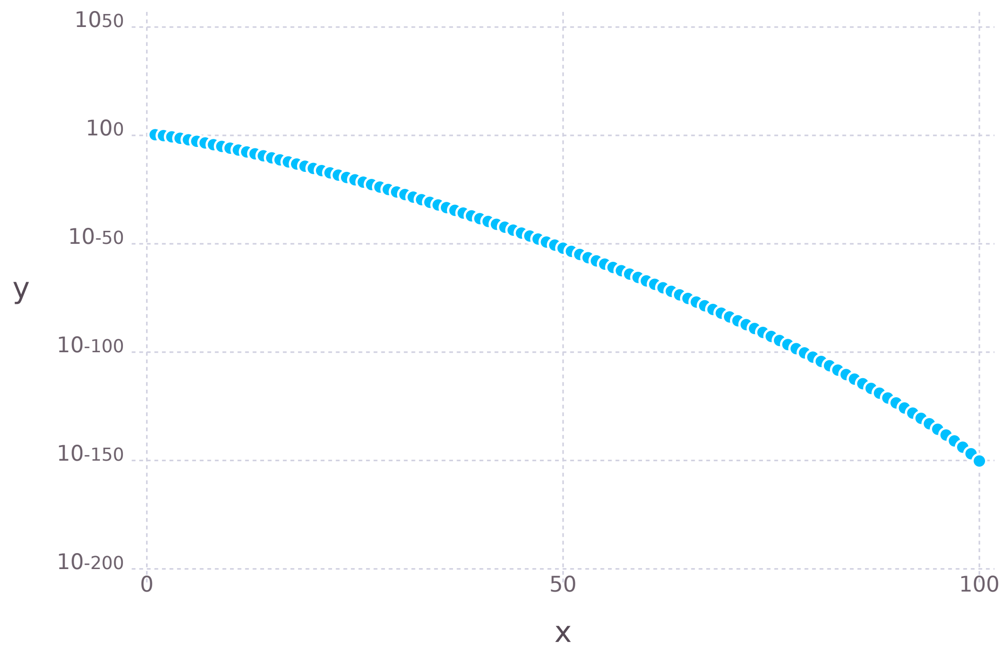
```
Out[26]: myRRD (generic function with 1 method)

In [27]: X,D,Y=myRRD(C);

In [28]: # Check
         norm((X*diagm(D)*Y')-full(C))

Out[28]: 1.2340699192861446e-16

In [29]: # Is this RRD? here X=Y
         cond(X), cond(Y)

Out[29]: (72.24644120521842,72.24644120521842)

In [30]: # Algorithm from Fact 3
         function myRRDSVD(X,D,Y)
             Q,R,p=qr(X*diagm(D),Val{true},thin=true)
             W=R[:,p]*Y'
             V,σ,U1=myJacobiR(W')
             U=Q*U1
             U,σ,V
         end

Out[30]: myRRDSVD (generic function with 1 method)

In [31]: U,σ,V=myRRDSVD(X,D,Y);

In [32]: # Check residual and orthogonality
         norm(full(C)*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)

Out[32]: (1.233471092882553e-15,2.604664737893255e-15,9.330915014096382e-15)

In [33]: # Observe the difference!!
         [sort(σ) sort(svdvals(C)) sort(eigvals(full(C)))]

Out[33]: 100x3 Array{Float64,2}:
          5.7797e-151    1.16516e-19  -5.8014e-16
          1.29735e-147   1.59193e-19  -4.96923e-16
          1.44439e-144   3.7406e-19   -3.4727e-16
          1.06342e-141   4.40985e-19  -2.52902e-16
          5.82434e-139   6.052e-19    -1.82281e-16
          2.5311e-136    6.33403e-19  -1.43636e-16
          9.09071e-134   7.85808e-19  -1.29068e-16
          2.77536e-131   8.84661e-19  -1.04099e-16
          7.35195e-129   1.06447e-18  -7.86621e-17
          1.71656e-126   1.25096e-18  -7.16606e-17
          3.57648e-124   1.29959e-18  -6.29428e-17
          6.71629e-122   1.41074e-18  -5.68772e-17
          1.14619e-119   1.45684e-18  -4.60919e-17
          ⋮
          2.41265e-8     2.41265e-8    2.41265e-8
          1.78872e-7     1.78872e-7    1.78872e-7
```

```
1.26617e-6     1.26617e-6     1.26617e-6
8.53628e-6     8.53628e-6     8.53628e-6
5.46453e-5     5.46453e-5     5.46453e-5
0.000330868    0.000330868    0.000330868
0.00188506     0.00188506     0.00188506
0.0100318      0.0100318      0.0100318
0.0492923      0.0492923      0.0492923
0.218596       0.218596       0.218596
0.821446       0.821446       0.821446
2.1827         2.1827         2.1827
```

In [35]:
```
# Plot the eigenvalues (singular values) and left singular vectors
using Gadfly
```

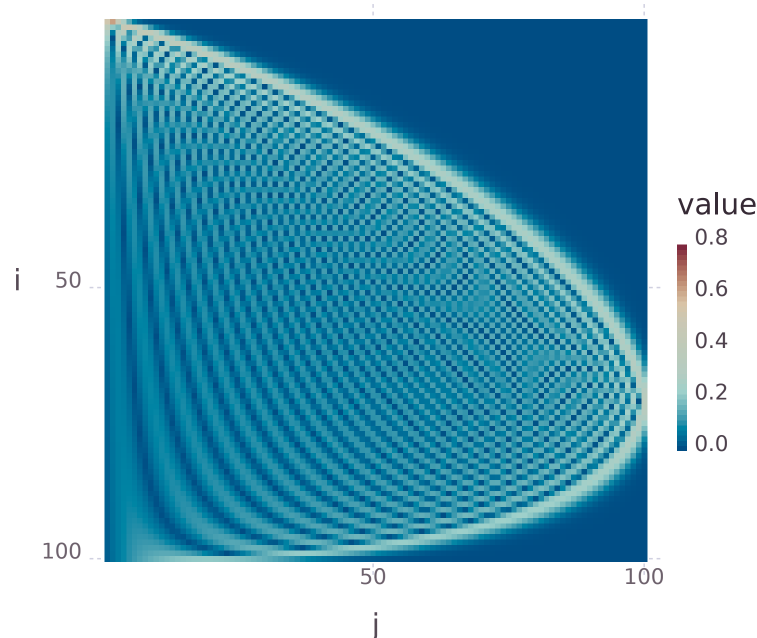
In [36]:
```
plot(x=collect(1:length(σ)),y=σ,Scale.y_log10)
```

Out[36]:



In [37]:
```
spy(abs(U))
```

Out[37]:



10.4 Symmetric arrowhead and DPR1 matrices

For more details, see N. Jakovčević Stor, I. Slapničar and J. Barlow, Accurate eigenvalue decomposition of real symmetric arrowhead matrices and applications and N. Jakovčević Stor, I. Slapničar and J. Barlow, Forward stable eigenvalue decomposition of rank-one modifications of diagonal matrices.

10.4.1 Definitions

An **arrowhead matrix** is a real symmetric matrix of order n of the form $A=\begin{bmatrix} D & z \\ z^T & \alpha \end{bmatrix}$, where $D=\operatorname{diag}(d_1,d_2,\ldots,d_{n-1})$, $z=\begin{bmatrix}\zeta_1 & \zeta_2 & \cdots & \zeta_{n-1}\end{bmatrix}^T$ is a vector, and α is a scalar.
An arrowhead matrix is **irreducible** if $\zeta_i\neq 0$ for all i and $d_i\neq d_j$ for all $i\neq j$.
A **diagonal-plus-rank-one matrix** (DPR1 matrix) is a real symmetric matrix of order n of the form $A=D+\rho zz^T$, where $D=\operatorname{diag}(d_1,d_2,\ldots,d_n)$, $z=\begin{bmatrix}\zeta_1 & \zeta_2 & \cdots & \zeta_n\end{bmatrix}^T$ is a vector, and $\rho\neq 0$ is a scalar.
A DPR1 matrix is **irreducible** if $\zeta_i\neq 0$ for all i and $d_i\neq d_j$ for all $i\neq j$.

10.4.2 Facts on arrowhead matrices

Let A be an arrowhead matrix of order n and let $A=U\Lambda U^T$ be its EVD.

1. If d_i and λ_i are nonincreasingy ordered, the Cauchy Interlace Theorem implies

$$\lambda_1\geq d_1\geq \lambda_2\geq d_2\geq\cdots\geq d_{n-2}\geq\lambda_{n-1}\geq d_{n-1}\geq\lambda_n.$$

2. If $\zeta_i = 0$ for some i, then d_i is an eigenvalue whose corresponding eigenvector is the i-th unit vector, and we can reduce the size of the problem by deleting the i-th row and column of the matrix. If $d_i = d_j$, then d_i is an eigenvalue of A (this follows from the interlacing property) and we can reduce the size of the problem by annihilating ζ_j with a Givens rotation in the (i, j)-plane.

3. If A is irreducible, the interlacing property holds with strict inequalities.

4. The eigenvalues of A are the zeros of the **Pick function**

$$f(\lambda) = \alpha - \lambda - \sum_{i=1}^{n-1} \frac{\zeta_i^2}{d_i - \lambda} = \alpha - \lambda - z^T (D - \lambda I)^{-1} z,$$

and the corresponding eigenvectors are

$$U_{:,i} = \frac{x_i}{\|x_i\|_2}, \quad x_i = \begin{bmatrix} (D - \lambda_i I)^{-1} z \\ -1 \end{bmatrix}, \quad i = 1, \ldots, n.$$

5. Let A be irreducible and nonsingular. If $d_i \neq 0$ for all i, then A^{-1} is a DPR1 matrix

$$A^{-1} = \begin{bmatrix} D^{-1} & \\ & 0 \end{bmatrix} + \rho u u^T,$$

where $u = \begin{bmatrix} z^T D^{-1} \\ -1 \end{bmatrix}$, and $\rho = \dfrac{1}{\alpha - z^T D^{-1} z}$. If $d_i = 0$, then A^{-1} is a permuted arrowhead matrix,

$$A^{-1} \equiv \begin{bmatrix} D_1 & 0 & 0 & z_1 \\ 0 & 0 & 0 & \zeta_i \\ 0 & 0 & D_2 & z_2 \\ z_1^T & \zeta_i & z_2^T & \alpha \end{bmatrix}^{-1} = \begin{bmatrix} D_1^{-1} & w_1 & 0 & 0 \\ w_1^T & b & w_2^T & 1/\zeta_i \\ 0 & w_2 & D_2^{-1} & 0 \\ 0 & 1/\zeta_i & 0 & 0 \end{bmatrix},$$

where $w_1 = -D_1^{-1} z_1 \dfrac{1}{\zeta_i}$, $w_2 = -D_2^{-1} z_2 \dfrac{1}{\zeta_i}$, and $b = \dfrac{1}{\zeta_i^2} \left(-\alpha + z_1^T D_1^{-1} z_1 + z_2^T D_2^{-1} z_2 \right)$.

6. The algorithm based on the following approach computes all eigenvalues and *all components* of the corresponding eigenvectors in a forward stable manner to almost full accuracy in $O(n)$ operations per eigenpair:

   1. Shift the irreducible A to d_i which is closer to λ_i (one step of bisection on $f(\lambda)$).
   2. Invert the shifted matrix.
   3. Compute the absolutely largest eigenvalue of the inverted shifted matrix and the corresponding eigenvector.

7. The algorithm is implemented in the package Arrowhead.jl. In certain cases, b or ρ need to be computed with extended precision for which the package DoubleDouble.jl is used.

10.4.3 Example - Random arrowhead matrix

In [38]:
```
# Pkg.add("Arrowhead"); Pkg.checkout("Arrowhead")
using Arrowhead
```

In [39]:
```
whos(Arrowhead)
```

```
   Arrowhead    267 KB     Module
 GenHalfArrow   687 bytes  Function
  GenSymArrow   577 bytes  Function
   GenSymDPR1   531 bytes  Function
    HalfArrow   180 bytes  DataType
     SymArrow   204 bytes  DataType
      SymDPR1   192 bytes  DataType
       bisect  6870 bytes  Function
          eig    67 KB     Function
          inv   135 KB     Function
    rootsWDK   1673 bytes  Function
     rootsah     15 KB     Function
          svd    33 KB     Function
          tdc  4054 bytes  Function
```

```
In [40]: methods(GenSymArrow)

Out[40]: # 1 method for generic function "GenSymArrow":
         GenSymArrow(n::Integer, i::Integer) at /home/slap/.julia/v0.4/Arrowhead/src/arrowhe

In [41]: n=10
         A=GenSymArrow(n,n)

Out[41]: 10x10 Arrowhead.SymArrow{Float64}:
          0.73791   0.0       0.0       0.0       ...  0.0       0.0       0.209041
          0.0       0.700993  0.0       0.0            0.0       0.0       0.744403
          0.0       0.0       0.322868  0.0            0.0       0.0       0.749886
          0.0       0.0       0.0       0.788659       0.0       0.0       0.837603
          0.0       0.0       0.0       0.0            0.0       0.0       0.974503
          0.0       0.0       0.0       0.0       ...  0.0       0.0       0.323502
          0.0       0.0       0.0       0.0            0.0       0.0       0.285448
          0.0       0.0       0.0       0.0            0.809975  0.0       0.146851
          0.0       0.0       0.0       0.0            0.0       0.833166  0.53399
          0.209041  0.744403  0.749886  0.837603       0.146851  0.53399   0.292361

In [42]: # Elements of the type SymArrow
         A.D, A.z, A.a, A.i

Out[42]: ([0.7379098714188161,0.700992971367602,0.3228677579964092,0.7886590867273988,0.487

In [43]: tols=[1e2,1e2,1e2,1e2,1e2]
         U,λ=eig(A,tols)
         norm(full(A)*U-U*diagm(λ)), norm(U'*U-I)

Out[43]: (1.0064326358048316e-15,3.9471334251957593e-16)

In [44]: # Timings - notice the O(n^2)
         @time eig(GenSymArrow(1000,1000),tols);
         @time eig(GenSymArrow(2000,2000),tols);

  0.363242 seconds (1.06 M allocations: 119.107 MB, 5.97% gc time)
  1.276176 seconds (4.14 M allocations: 472.270 MB, 7.81% gc time)
```

10.4.4 Example - Numerically demanding matrix

```
In [45]: A=SymArrow( [ 1e10+1.0/3.0, 4.0, 3.0, 2.0, 1.0 ], [ 1e10 - 1.0/3.0, 1.0, 1.0, 1.0,
```

```
Out[45]: 6x6 Arrowhead.SymArrow{Float64}:
          1.0e10  0.0  0.0  0.0  0.0  1.0e10
          0.0     4.0  0.0  0.0  0.0  1.0
          0.0     0.0  3.0  0.0  0.0  1.0
          0.0     0.0  0.0  2.0  0.0  1.0
          0.0     0.0  0.0  0.0  1.0  1.0
          1.0e10  1.0  1.0  1.0  1.0  1.0e10
```

```
In [46]: U,λ=eig(A,tols);
         println([sort(λ) sort(eigvals(full(A)))])
```

```
[-0.3481422590562395 -0.34814214957825623
 1.2618505092343664 1.2618505256908337
 2.2232515665900348 2.2232515792896965
 3.1883186353364037 3.1883186449586667
 4.174722501468362 4.1747225102436145
 1.999999999983333e10 1.9999999999833336e10]
```

10.4.5 Facts on DPR1 matrices

The properties of DPR1 matrices are very similar to those of arrowhead matrices. Let A be a DPR1 matrix of order n and let $A = U\Lambda U^T$ be its EVD.

1. If d_i and λ_i are nonincreasingy ordered and $\rho > 0$, then

   $$\lambda_1 \geq d_1 \geq \lambda_2 \geq d_2 \geq \cdots \geq d_{n-2} \geq \lambda_{n-1} \geq d_{n-1} \geq \lambda_n \geq d_n.$$

   If A is irreducible, the inequalities are strict.

2. Facts 2 on arrowhead matrices holds.

3. The eigenvalues of A are the zeros of the **secular equation**

   $$f(\lambda) = 1 + \rho \sum_{i=1}^{n} \frac{\zeta_i^2}{d_i - \lambda} = 1 + \rho z^T (D - \lambda I)^{-1} z = 0,$$

   and the corresponding eigenvectors are

   $$U_{:,i} = \frac{x_i}{\|x_i\|_2}, \quad x_i = (D - \lambda_i I)^{-1} z.$$

4. Let A be irreducible and nonsingular. If $d_i \neq 0$ for all i, then

   $$A^{-1} = D^{-1} + \gamma u u^T, \quad u = D^{-1} z, \quad \gamma = -\frac{\rho}{1 + \rho z^T D^{-1} z},$$

   is also a DPR1 matrix. If $d_i = 0$, then A^{-1} is a permuted arrowhead matrix,

   $$A^{-1} \equiv \left(\begin{bmatrix} D_1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & D_2 \end{bmatrix} + \rho \begin{bmatrix} z_1 \\ \zeta_i \\ z_2 \end{bmatrix} \begin{bmatrix} z_1^T & \zeta_i & z_2^T \end{bmatrix} \right)^{-1} = \begin{bmatrix} D_1^{-1} & w_1 & 0 \\ w_1^T & b & w_2^T \\ 0 & w_2 & D_2^{-1} \end{bmatrix},$$

   where $w_1 = -D_1^{-1} z_1 \dfrac{1}{\zeta_i}$, $w_2 = -D_2^{-1} z_2 \dfrac{1}{\zeta_i}$, and $b = \dfrac{1}{\zeta_i^2} \left(\dfrac{1}{\rho} + z_1^T D_1^{-1} z_1 + z_2^T D_2^{-1} z_2 \right)$.

5. The algorithm based on the same approach as above, computes all eigenvalues and all components of the corresponding eigenvectors in a forward stable manner to almost full accuracy in $O(n)$ operations per eigenpair. The algorithm is implemented in the package `Arrowhead.jl`. In certain cases, b or γ need to be computed with extended precision.

10.4.6 Example - Random DPR1 matrix

```
In [47]: n=10
         A=GenSymDPR1(n)

Out[47]: 10x10 Arrowhead.SymDPR1{Float64}:
          0.610202    0.0117837  0.0192761  ...  0.00602551  0.00348987  0.00551833
          0.0117837   0.18538    0.198286        0.061982    0.035899    0.0567648
          0.0192761   0.198286   0.632466        0.101392    0.0587244   0.0928573
          0.0139178   0.143167   0.234197        0.0732073   0.0424005   0.0670453
          0.0113843   0.117106   0.191564        0.0598809   0.034682    0.0548406
          0.0158796   0.163347   0.267207   ...  0.083526    0.0483769   0.0764955
          0.00419316  0.0431334  0.0705586       0.0220559   0.0127744   0.0201994
          0.00602551  0.061982   0.101392        0.684669    0.0183566   0.0290262
          0.00348987  0.035899   0.0587244       0.0183566   0.338348    0.0168115
          0.00551833  0.0567648  0.0928573       0.0290262   0.0168115   0.103211

In [48]: # Elements of the type SymDPR1
         A.D, A.u, A.r

Out[48]: ([0.6090565674680575,0.06416503772960547,0.3081054529139915,0.249508639844354,0.768

In [49]: U,λ=eig(A,tols)
         norm(full(A)*U-U*diagm(λ)), norm(U'*U-I)

Out[49]: (3.141879657198308e-16,4.854740367641191e-16)
```

10.4.7 Example - Numerically demanding matrix

```
In [50]: A=SymDPR1( [ 10.0/3.0, 2.0+1e-7, 2.0-1e-7, 1.0 ], [ 2.0, 1e-7, 1e-7, 2.0], 1.0 )
         A = SymDPR1( [ 1e10, 5.0, 4e-3, 0.0, -4e-3,-5.0 ], [ 1e10, 1.0, 1.0, 1e-7, 1.0,1.0

Out[50]: 6x6 Arrowhead.SymDPR1{Float64}:
             1.0e20  1.0e10  1.0e10  1000.0       1.0e10   1.0e10
             1.0e10  6.0     1.0         1.0e-7   1.0      1.0
             1.0e10  1.0     1.004       1.0e-7   1.0      1.0
          1000.0     1.0e-7  1.0e-7      1.0e-14  1.0e-7   1.0e-7
             1.0e10  1.0     1.0         1.0e-7   0.996    1.0
             1.0e10  1.0     1.0         1.0e-7   1.0     -4.0

In [51]: U,λ=eig(A,tols)
         norm(full(A)*U-U*diagm(λ)), norm(U'*U-I), println([sort(λ) sort(eigvals(full(A)))]

Remedy 3
[-4.9999999999 -4.999999999900006
 -0.003999999900000001 -5.742148820240562e-14
```

```
 9.999999998999997e-25 5.6271169015955564e-14
 0.004000000100000001 0.0040000000999998505
 5.0000000001 5.0000000001
 1.0000000001e20 1.0000000001000002e20]
```

```
Out[51]: (3.0381820395446957e-6,2.2204460858891437e-16,nothing)
```

```
In [ ]:
```

11 Updating the SVD

In many applications which are based on the SVD, arrival of new data requires SVD of the new matrix. Instead of computing from scratch, existing SVD can be updated.

11.1 Prerequisites

The reader should be familiar with concepts of singular values and singular vectors, related perturbation theory, and algorithms.

11.2 Competences

The reader should be able to recognise applications where SVD updating can be sucessfully applied and apply it.

11.3 Facts

For more details see M. Gu and S. C. Eisenstat, A Stable and Fast Algorithm for Updating the Singular Value Decomposition and M. Brand, Fast low-rank modifications of the thin singular value decomposition and the references therein.

1. Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and $\operatorname{rank}(A) = n$, and let $A = U\Sigma V^T$ be its SVD. Let $a \in \mathbb{R}^n$ be a vector, and let $\tilde A = \begin{bmatrix} A \\ a^T \end{bmatrix}$. Then
$$\begin{bmatrix} A \\ a^T \end{bmatrix} = \begin{bmatrix} U & \\ & 1 \end{bmatrix} \begin{bmatrix} \Sigma \\ a^T V \end{bmatrix} V^T.$$
Let $\begin{bmatrix} \Sigma \\ a^T V \end{bmatrix} = \bar U \bar\Sigma \bar V^T$ be the SVD of the half-arrowhead matrix. *This SVD can be computed in $O(n^2)$ operations.* Then
$$\begin{bmatrix} A \\ a^T \end{bmatrix} = \begin{bmatrix} U & \\ & 1 \end{bmatrix} \bar U \bar\Sigma \bar V^T V^T \equiv \tilde U \bar\Sigma \tilde V^T$$
is the SVD of $\tilde A$.

2. Direct computation of $\tilde U$ and $\tilde V$ requires $O(mn^2)$ and $O(n^3)$ operations. However, these multiplications can be performed using Fast Multipole Method. This is not (yet) implemented in Julia and is "not for the timid" (quote by Steven G. Johnson).

3. If $m < n$ and $\operatorname{rank}(A) = n$, then
$$\begin{bmatrix} A \\ a^T \end{bmatrix} = \begin{bmatrix} U & \\ & 1 \end{bmatrix} \begin{bmatrix} \Sigma & 0 \\ a^T V & \beta \end{bmatrix} \begin{bmatrix} V^T \\ v^T \end{bmatrix},$$
where $\beta = \sqrt{\|a\|_2^2 - \|V^T a\|_2^2}$ and $v = (I - VV^T)a$. Notice that $V^T v = 0$ by construction. Let $\begin{bmatrix} \Sigma & 0 \\ a^T V & \beta \end{bmatrix} = \bar U \bar\Sigma \bar V^T$ be the SVD of the half-arrowhead matrix. Then
$$\begin{bmatrix} A \\ a^T \end{bmatrix} = \begin{bmatrix} U & \\ & 1 \end{bmatrix} \bar U \bar\Sigma \bar V^T \begin{bmatrix} V^T \\ v^T \end{bmatrix} \equiv \tilde U \bar\Sigma \tilde V^T$$
is the SVD of $\tilde A$.

4. Adding a column a to A is equivalent to adding a row a^T to A^T.

5. If $\operatorname{rank}(A) < \min\{m, n\}$ or if we are using SVD approximation of rank r, and if we want to keep the rank of the approximation (this is the common case in practice), then the formulas in Fact 1 hold approximately. More precisely, the updated rank r approximation is **not** what we would get by computing the approximation of rank r of the updated matrix, but is sufficient in many applications.

11.3.1 Example - Adding row to a tall matrix

If $m >= n$, adding row does not increase the size of Σ.

```
In [1]: using Arrowhead
```

```
In [2]: function mySVDaddrow{T}(svdA::Tuple,a::Vector{T})
            # Create the transposed half-arrowhead
            m,r,n=size(svdA[1],1),length(svdA[2]),size(svdA[3],1)
            b=svdA[3]'*a
            if m>=n || r<m
                M=HalfArrow(svdA[2],b)
            else
                β=sqrt(vecnorm(a)^2-vecnorm(b)^2)
                M=HalfArrow(svdA[2],[b;β])
            end
            tols=[1e2,1e2,1e2,1e2]
            U,σ,V=svd(M,tols)
            # Return the updated SVD
            if m>=n || r<m
                return [svdA[1] zeros(T,m); zeros(T,1,r) one(T)]*V, σ, svdA[3]*U
            else
                # Need one more row of svdA[3] - v is orthogonal projection
                v=a-svdA[3]*b
                v=v/norm(v)
                return [svdA[1] zeros(T,m); zeros(T,1,r) one(T)]*V, σ, [svdA[3] v]*U
            end
        end
```

```
Out[2]: mySVDaddrow (generic function with 1 method)
```

```
In [3]: A=rand(10,6)
        a=rand(6)
```

```
Out[3]: 6-element Array{Float64,1}:
         0.221399
         0.768015
         0.434068
         0.669569
         0.349026
         0.937966
```

```
In [4]: svdA=svd(A)
```

```
Out[4]: (
        10x6 Array{Float64,2}:
         -0.243555   0.762451   -0.161994   -0.259465   -0.138777    0.375221
         -0.246189  -0.0119037   0.498387   -0.0175853  -0.322755    0.195748
         -0.317533  -0.361891    0.043925   -0.494296   -0.0696735   0.359471
         -0.178403  -0.184867   -0.427682   -0.0953758   0.473642    0.29752
         -0.354006   0.178985    0.0179635   0.194722    0.551791    0.0198701
         -0.422596  -0.216432   -0.0906625   0.617458   -0.0657186   0.126564
         -0.387754   0.118349   -0.130384   -0.309193    0.0544702  -0.753264
         -0.239722   0.217977    0.578327    0.173827    0.229692   -0.0575153
         -0.395297  -0.325427    0.0996438  -0.229156   -0.118191   -0.101937
         -0.281834   0.0841711  -0.41303     0.285384   -0.518548   -0.0750654,

        [4.0669223594657655,1.1197418382358004,0.9946878371522861,0.8485590415477365,0.67420
        6x6 Array{Float64,2}:
         -0.372081   0.689015   -0.227543   0.0961875   0.392363   -0.414533
         -0.481432  -0.625971   -0.467782  -0.216512    0.316648   -0.102081
         -0.428354   0.330977   -0.221659  -0.334495   -0.341991    0.654973
         -0.377118  -0.110294   -0.100398   0.527336   -0.679148   -0.310182
         -0.460709  -0.0610131   0.758855  -0.389289   -0.0417284  -0.234257
         -0.30302   -0.0893633   0.307135   0.634314    0.407239    0.487507)

In [5]: typeof(svdA)

Out[5]: Tuple{Array{Float64,2},Array{Float64,1},Array{Float64,2}}

In [6]: U,σ,V=mySVDaddrow(svdA,a)

Remedy 3


Out[6]: (
        11x6 Array{Float64,2}:
         -0.223588   0.741827   -0.190412    -0.0927312    0.0730524  -0.487543
         -0.233061   0.0402103   0.501492     0.00365628   0.315254   -0.187775
         -0.300305  -0.151736    0.102506     0.571258     0.0186496  -0.391427
         -0.171012  -0.141164   -0.40899      0.179551    -0.47827    -0.205638
         -0.336224   0.142568   -0.00437316  -0.220858    -0.521894    0.101039
         -0.407184  -0.283662   -0.112532    -0.347054     0.150506    0.122691
         -0.362356   0.265978   -0.105999     0.305316    -0.058596    0.617229
         -0.226749   0.18621     0.552408    -0.270214    -0.208507    0.114133
         -0.374709  -0.153561    0.142184     0.377957     0.114103    0.0938557
         -0.268033   0.0521039  -0.428354    -0.159883     0.553611    0.120539
         -0.32008   -0.415477   -0.0411357   -0.359083    -0.068915   -0.298231 ,

        [4.286621653219806,1.204402274736068,0.9951041265454403,0.9577533795504506,0.677478
        6x6 Array{Float64,2}:
         -0.349721   0.642482  -0.268356  -0.281701  -0.372344    0.418213
         -0.491016  -0.493593  -0.412129   0.480405  -0.313598    0.127621
         -0.416328   0.381662  -0.218476   0.150065   0.269639   -0.733518
         -0.389809  -0.208675  -0.124604  -0.328395   0.744455    0.356258
         -0.439508   0.148307   0.794757   0.352512   0.0190706   0.169033
         -0.343731  -0.363837   0.251474  -0.659718  -0.36842    -0.339515)
```

```
In [7]: # Check the residual and orthogonality
        norm([A;a']*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Out[7]: (5.589014826080515e-15,1.348466742720976e-15,1.1244364582318845e-15)
```

11.3.2 Example - Adding row to a flat matrix

```
In [8]: # Now flat matrix
        A=rand(6,10)
        a=rand(10)
        svdA=svd(A)
```

```
Out[8]: (
        6x6 Array{Float64,2}:
         -0.492899  -0.632294    0.219741   -0.040012    0.250835   -0.494419
         -0.281561   0.498295    0.78851    -0.224182    0.0107329   0.0174794
         -0.463948   0.559196   -0.528127   -0.047077    0.258021   -0.352622
         -0.53031   -0.138779   -0.188737   -0.31602    -0.689295    0.29815
         -0.242783  -0.132008   -0.0781563  -0.17855     0.62207     0.706167
         -0.34979    0.0502126   0.0965287   0.902317   -0.0910671   0.208178 ,

        [4.030811316421142,1.1737792968182705,0.9750642580493722,0.8398159144651589,0.598794
        10x6 Array{Float64,2}:
         -0.315261  -0.275586     0.062729   -0.147609   -0.0134245  -0.0914208
         -0.179688   0.104346     0.569515    0.178287    0.422191    0.0625051
         -0.349289  -0.00768558   0.239263   -0.677574   -0.232717   -0.14997
         -0.222914  -0.456098     0.186059    0.0425505   0.48718     0.362133
         -0.409889  -0.227428    -0.12584     0.011468   -0.0518926  -0.03536
         -0.22917    0.302136    -0.323826    0.138017    0.528318   -0.573908
         -0.4069     0.332244    -0.248962    0.305025   -0.202882    0.583587
         -0.332932   0.0543725   -0.469054   -0.316218    0.177874    0.133485
         -0.245067   0.60728      0.40996    -0.034311   -0.0930448  -0.018225
         -0.373674  -0.282734     0.0953569   0.521648   -0.408415   -0.380472 )
```

```
In [9]: U,σ,V=mySVDaddrow(svdA,a)
        norm([A;a']*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Out[9]: (8.010550988849256e-15,1.6726991807756699e-15,2.9517375933747436e-15)
```

11.3.3 Example - Adding columns

This can be viewed as adding rows to the transposed matrix, an elegant one-liner in Julia.

```
In [10]: function mySVDaddcol{T}(svdA::Tuple,a::Vector{T})
             reverse(mySVDaddrow(reverse(svdA),a))
         end
```

```
Out[10]: mySVDaddcol (generic function with 1 method)
```

```
In [11]: # Tall matrix
         A=rand(10,6)
         a=rand(10)
```

```
svdA=svd(A)
U,σ,V=mySVDaddcol(svdA,a)
norm([A a]*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Remedy 3
```

```
Out[11]: (3.136901403015872e-15,3.2608700902653993e-15,1.753752094124507e-15)
```

```
In [12]: # Flat matrix
         A=rand(6,10)
         a=rand(6)
         svdA=svd(A)
         U,σ,V=mySVDaddcol(svdA,a)
         norm([A a]*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Out[12]: (1.883988810118734e-15,1.0269170176271777e-15,9.120667403637087e-16)
```

```
In [13]: # Square matrix
         A=rand(10,10)
         a=rand(10);
         svdA=svd(A);
```

```
In [14]: U,σ,V=mySVDaddrow(svdA,a)
         norm([A;a']*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Remedy 3
```

```
Out[14]: (3.5713454174155424e-14,1.4196141008832963e-15,6.4889861569114546e-15)
```

```
In [15]: U,σ,V=mySVDaddcol(svdA,a)
         norm([A a]*V-U*diagm(σ)), norm(U'*U-I), norm(V'*V-I)
```

```
Remedy 3
```

```
Out[15]: (3.533535952914256e-14,1.4269971076026336e-15,1.1828442053492742e-15)
```

11.3.4 Example - Updating a low rank approximation

```
In [16]: # Adding row to a tall matrix
         A=rand(10,6)
         svdA=svd(A)
         a=rand(6)
         # Rank of the approximation
         r=4
```

```
Out[16]: 4
```

```
In [17]: svdAr=(svdA[1][:,1:r], svdA[2][1:r],svdA[3][:,1:r])
         U,σ,V=mySVDaddrow(svdAr,a)
         norm([A;a']-U*diagm(σ)*V'), svdvals([A;a']), σ
```

```
Out[17]: (0.42361407360119013,[4.144646535891786,1.339670070904022,1.0412845751261728,0.668

In [18]: # Adding row to a flat matrix
         A=rand(6,10)
         svdA=svd(A)
         a=rand(10)
         # Rank of the approximation
         r=4

Out[18]: 4

In [19]: svdAr=(svdA[1][:,1:r], svdA[2][1:r],svdA[3][:,1:r])
         U,σ,V=mySVDaddrow(svdAr,a)
         norm([A;a']-U*diagm(σ)*V'), svdvals([A;a']), σ

Out[19]: (0.8608726378200711,[4.089986400967123,1.2934152855408454,1.085652181293417,0.7867

In [ ]:
```

12 Tutorial 2 - Examples in Eigenvalue Decomposition

12.1 Assignment 1

Using the file dl-matrixmarket.jl from the package MatrixMarket.jl (copy the file to your notebook), download two randomly chosen matrices (make sure that both matrices are real and at least one is real symmetric).

For each matrix: * inspect the properties of the matrix (size, symmetry, condition number, sparsity, structure, ...), * plot the matrix using the command `spy()` from the package `Gadfly.jl`, * compute the eigenvalue decomposition with appropriate methods, and * assess the accuracy of the decomposition.

Hints

1. In Windows, you may need to prepend the `http://` to the address in the `download()` command.

2. To plot the matrix `A`, use the following commands:

   ```
   myplot=spy(A)
   draw(PNG(12cm,12cm),myplot)
   ```

   To see only the structure, use

   ```
   myplot=spy(map(Int64,A.!=0.0))
   ```

   For larger matrices, plotting takes a while.

12.2 Assignment 2

Choose three matrices from the package MatrixDepot.jl: * one from the class "eigen", * one which is sparse and symmetric, and * one which is ill-conditioned and symmetric,

and analyse them as described in the Assignment 1.

12.3 Assignment 3*

Learn about Google's Page Rank algorithm and explain it as an eigenvalue problem. Implement the algorithm and analyse Stanford web graph.

Suggested readings:

- A. N. Langville and C. D. Meyer, Information Retrieval and Web Search
- A. N. Langville and C. D. Meyer, Deeper Inside PageRank
- C, Moler, Google PageRank
- P. Dreher et al., PageRank Pipeline Benchmark

```
In [ ]:
```

13 Tutorial 3 - Examples in Singular Value Decomposition

13.1 Assignment 1

Using the file dl-matrixmarket.jl from the package MatrixMarket.jl (copy the file to your notebook), download two randomly chosen matrices.

For each matrix: * inspect the properties of the matrix (size, symmetry, condition number, sparsity, structure, scaled condition number, ...), * plot the matrix using the command `spy()` from the package `Gadfly.jl`, * compute the singular value decomposition, and * assess its accuracy.

Hints

1. In Windows, you may need to prepend the `http://` to the address in the `download()` command.
2. To plot the matrix `A`, use the following commands:

   ```
   myplot=spy(A)
   draw(PNG(12cm,12cm),myplot)
   ```

   To see only the structure, use

   ```
   myplot=spy(map(Int64,A.!=0.0))
   ```

   For larger matrices, plotting takes a while.

13.2 Assignment 2

Choose an image from the package TestImages.jl or find an image elsewhere.

Compute low-rank aproximations of the image and display them using `@manipulate`.

13.3 Assignment 3

Write a wrapper for DGESVJ similar to those from the file lapack.jl.

Test the function on a strongly scaled matrix.

For larger matrices, compare timings with `svd()`.

```
In [ ]:
```

14 Tutorial 4 - Examples in SVD Updating

14.1 Assignment 1*

The Netflix Recommendation Engine is based on the principle of a low-rank (truncated) SVD approximation of the (large and sparse) Movie $\times$ Users matrix, and the SVD updating.

Explain the mathematics behind the engine using explanations from the packages IncrementalSVD.jl, and RecSys.jl (see also A parallel recommendation engine in Julia), and the literature on the Internet.

Explain the relation to SVD, truncated SVD, and SVD updating.

Try the packages.

14.2 Assignment 2**

Implement the matrix multiplication for SVD updating using Fast Multipole Method as described in M. Gu and S. C. Eisenstat, A Stable and Fast Algorithm for Updating the Singular Value Decomposition.

```
In [ ]:
```