L7 Algorithms for Structured Matrices

Ivan Slapničar

May 2, 2018

1 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.

1.1 Prerequisites

The reader should be familiar with concepts of eigenvalues and eigen vectors, singular values and singular vectors, related perturbation theory, and algorithms.

1.2 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.

1.3 Rank revealing decompositions

For more details, see [Drm14] 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.

References

[Drm14] Z. Drmač, Computing Eigenvalues and Singular Values to High Relative Accuracy, in L. Hogben, ed., 'Handbook of Linear Algebra', pp. 59.1-59.21, CRC Press, Boca Raton, 2014.

1.3.1 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

$$\min\{|\theta|, 1/|\theta|\}\sigma_j \leq \tilde{\sigma}_j \leq \max\{|\theta|, 1/|\theta|\}\sigma_j, \quad \forall j.$$

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.

1.3.2 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

$$\begin{aligned} |D_{jj} - \hat{D}_{jj}| &\leq O(\varepsilon)|D_{jj}|, \\ \|X - \hat{X}\| &\leq O(\varepsilon)\|X\|, \\ \|Y - \hat{Y}\| &\leq O(\varepsilon)\|Y\|. \end{aligned}$$

   The following algorithm computes the EVD of A with high relative accuracy:

   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 symmetric 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):

$$\begin{aligned}
D_{11} &= A_{11}, \\
D_{jj} &= \frac{\det(A_{1:j,1:j})}{\det(A_{1:j-1,1:j-1})}, \quad j = 2, \ldots, n, \\
L_{jj} &= 1, \\
L_{ij} &= \frac{\det(A_{[1,2,\ldots,j-1,i],[1:j]})}{\det(A_{1:j,1:j})}, \quad j < i, \\
R_{jj} &= 1, \\
R_{ji} &= \frac{\det(A_{[1,2,\ldots,j],[1,2,\ldots,j-1,i]})}{\det(A_{1:j,1:j})}, \quad i > j,
\end{aligned}$$

1.3.3 Example - Positive definite matrix

Let $A = DA_SD$ 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 $P^TAP = LL^T$ (*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
        s=srand(421)
        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])
        issymmetric(A), cond(As), cond(A)
```

```
Out[2]: (true, 1794.7674834322888, 4.7974903675033356e39)
```

```
In [3]: ?chol;
```

```
search: chol cholfact cholfact! searchsortedlast CachingPool chop chown chomp
```

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!
```

```
In [5]: p=sortperm(diag(A), rev=true)
        L=chol(A[p,p])
```

```
Out[5]: 20×20 UpperTriangular{Float64,Array{Float64,2}}:
         7.88685e8  -2.1039e7    15062.9       -290.504   ...  -3.44776e-11  -4.31924e-12
          ⋅          1.93457e8   20859.7       -111.39         -2.33051e-10  -6.10259e-13
          ⋅           ⋅              6.4718e5   108.871         5.37826e-11   1.45358e-11
          ⋅           ⋅               ⋅        1115.06          3.92092e-10  -5.36617e-12
          ⋅           ⋅               ⋅            ⋅           -1.01533e-10   1.1563e-11
          ⋅           ⋅               ⋅            ⋅      ...  -7.96891e-12  -1.02093e-11
          ⋅           ⋅               ⋅            ⋅           -1.50375e-10  -5.12743e-12
          ⋅           ⋅               ⋅            ⋅           -2.3185e-12   -7.73426e-12
          ⋅           ⋅               ⋅            ⋅           -3.32446e-10  -3.29658e-12
          ⋅           ⋅               ⋅            ⋅           -1.69797e-10   2.50981e-11
```

```
 .         .            .           .      ...  -1.61957e-10  -3.35054e-12
 .         .            .           .         -1.02456e-10   1.96947e-11
 .         .            .           .          2.31466e-10   1.39367e-11
 .         .            .           .         -5.43022e-10  -1.66566e-11
 .         .            .           .         -1.31544e-10   2.3234e-11
 .         .            .           .      ...   3.40047e-10   4.28435e-11
 .         .            .           .          1.73146e-10   1.99322e-13
 .         .            .           .         -1.15019e-10   2.62641e-12
 .         .            .           .          4.34304e-10  -6.22212e-12
 .         .            .           .            .            1.13816e-11
```

In [6]:
```
U,σ,V=myJacobiR(full(L'));
```

In [7]:
```
methods(myJacobiR)
```

Out[7]:
```
# 1 method for generic function "myJacobiR":
myJacobiR(A1::Array) in ModuleB at C:\Users\Ivan\Documents\Julia\GIAN-Applied-NLA-Course
```

In [8]:
```
λ=σ.^2
U₁=U[invperm(p),:]
λ
```

Out[8]:
```
20-element Array{Float64,1}:
    6.22495e17
    3.73973e16
    4.18842e11
    1.24604e6
    1.90729e5
 3078.7
  943.337
  303.624
    0.0324829
    0.00290981
    5.8684e-7
    3.7354e-10
    1.67425e-11
    8.05676e-13
    2.03386e-13
    6.09339e-16
    1.03927e-17
    6.24441e-19
    1.83052e-19
    1.29513e-22
```

In [10]:
```
# Due to large condition number, this is not
# as accurate as expected
C=U₁'*A*U₁
```

```
Out[10]: 20×20 Array{Float64,2}:
   6.22495e17   -0.18755      -0.0947112    ...   0.0610662     0.00999545
   1.58015       3.73973e16    0.00225244       -0.101356     -0.00714252
  -0.0970856     0.00240602    4.18842e11        5.46604e-5    9.76219e-6
   0.00285716    0.00673102    2.26899e-8        3.48398e-12  -7.42396e-13
   0.000884084   0.000311888   1.68231e-9       -9.12969e-13   3.20558e-13
  -1.07605e-5   -3.29096e-5    9.28433e-10  ...   8.72288e-23   2.8094e-12
   1.37333e-5    4.34536e-5   -8.43749e-10       2.61087e-15  -5.27696e-16
  -3.36893e-6    3.55029e-5    9.71202e-10       1.17891e-15   2.63558e-18
  -1.48646e-7   -6.53639e-7    2.38425e-5       -5.65083e-20   3.63978e-21
   1.82761e-7   -2.65804e-10   2.94551e-8       -5.69425e-22   3.54383e-24
  -6.0057e-9    -3.21174e-9    6.60497e-8   ...   8.62805e-24   1.53993e-24
   3.57949e-5    1.80388e-7   -2.56019e-9        2.68843e-24   4.80636e-25
  -0.0727832    -0.00182909   -5.45527e-10      -2.18273e-21  -8.19359e-22
 -21.8359        0.682556      8.98052e-11      -3.99197e-18  -4.80982e-19
  60.0456       12.7945       -3.17079e-11      -2.87859e-17  -1.47947e-18
 -10.0303        3.23745      -9.59354e-10  ...  -9.75826e-18  -7.79378e-19
   2.21946      -0.437917     -9.99335e-7        1.40446e-18   1.19253e-19
   0.372882     -0.185034      9.70223e-5        5.50729e-19   4.35884e-20
   0.0610662    -0.101356      5.46604e-5        4.70876e-19   2.16126e-20
   0.00999545   -0.00714252    9.76219e-6        2.16126e-20   1.8817e-21
```

```
In [11]: # Orthogonality
         vecnorm(U₁'*U₁-I)
```

```
Out[11]: 3.330032502108985e-15
```

```
In [12]: Dc=sqrt.(diag(C))
         Cs=map(Float64,[C[i,j]/(Dc[i]*Dc[j]) for i=1:n, j=1:n])
```

```
Out[12]: 20×20 Array{Float64,2}:
  1.0          -1.22922e-18  -1.85485e-16  ...   0.112792      0.292051
  1.03564e-17   1.0           1.79973e-17      -0.763794     -0.851444
 -1.90135e-16   1.92244e-17   1.0               0.123082      0.347734
  3.24415e-15   3.11813e-14   3.14082e-17       4.54838e-6   -1.53319e-5
  2.56577e-15   3.69292e-15   5.95212e-18      -3.04645e-6    1.69209e-5
 -2.458e-16    -3.06703e-15   2.58549e-17  ...   2.29099e-15   0.00116722
  5.66727e-16   7.31598e-15  -4.24478e-17       1.23879e-7   -3.96073e-7
 -2.4505e-16    1.0536e-14    8.61225e-17       9.85958e-8    3.48685e-9
 -1.04534e-15  -1.87538e-14   2.04409e-10      -4.56911e-10   4.65556e-10
  4.29421e-15  -2.54806e-17   8.4373e-13       -1.53833e-11   1.51448e-12
 -9.93656e-15  -2.16801e-14   1.33225e-10  ...   1.64134e-11   4.63411e-11
  2.34738e-9    4.82635e-11  -2.04682e-10       2.02711e-10   5.73288e-10
 -2.25452e-5   -2.31156e-6   -2.06007e-10      -7.77387e-7   -4.61625e-6
 -0.0308186     0.00393032    1.54521e-10      -0.00647805   -0.0123471
  0.164687      0.143169     -1.0602e-10       -0.0907759    -0.0738032
 -0.392102      0.51634      -4.57201e-8   ...  -0.438603     -0.554149
```

```
 0.581107     -0.467788     -0.00031898       0.422799      0.567898
 0.353662     -0.716006      0.112184         0.600579      0.751937
 0.112792     -0.763794      0.123082         1.0           0.726069
 0.292051     -0.851444      0.347734         0.726069      1.0
```

In [13]:
```
K=U₁*diagm(σ)
K'*K
```

Out[13]:
```
20×20 Array{Float64,2}:
  6.22495e17    0.469379     -7.84249e-5   ...   3.31146e-20   1.44176e-22
  0.469379      3.73973e16    4.68424e-6        -2.24242e-19  -4.20328e-22
 -7.84249e-5    4.68424e-6    4.18842e11         3.61356e-20   1.71664e-22
  4.10619e-9    3.88544e-8   -2.3475e-11         1.33536e-24  -7.5688e-27
  4.98538e-10   7.15053e-10   7.77502e-13       -8.94406e-25   8.35326e-27
 -7.70856e-13  -9.48588e-12   1.38482e-13  ...   8.04935e-35   5.76218e-25
  5.66117e-13   6.89736e-12   2.17019e-14        3.63697e-26  -1.95527e-28
 -1.14732e-13   3.23159e-12  -7.56646e-16        2.89467e-26   1.72134e-30
 -3.9013e-17   -6.07388e-16   6.6398e-12        -1.41535e-28   1.94731e-31
  1.18956e-17   3.35668e-19   2.45502e-15       -4.54705e-30   6.02133e-34
 -5.59609e-21  -1.28497e-20   7.81817e-17  ...  -1.15451e-36  -1.50231e-37
  8.76843e-19   1.80283e-20  -7.6457e-20        -6.89466e-35  -1.56744e-38
 -3.77462e-16  -3.87013e-17  -3.44906e-21       -1.37984e-35  -1.53721e-38
 -2.48418e-14   3.1681e-15    1.24554e-22       -6.23581e-35   3.63859e-38
  3.43221e-14   2.98376e-14  -2.20954e-23       -1.67284e-36   2.40574e-38
 -3.13816e-16   4.13249e-16  -3.65918e-23  ...  -5.16032e-36  -3.44068e-38
  9.06867e-18  -7.30022e-18  -4.97795e-21        1.62085e-36  -7.92831e-39
  3.73464e-19  -7.56095e-19   1.18466e-19       -1.90057e-35  -5.12707e-39
  3.31146e-20  -2.24242e-19   3.61356e-20        1.83052e-19   2.29808e-39
  1.44176e-22  -4.20328e-22   1.71664e-22        2.29808e-39   1.29513e-22
```

In [14]:
```
# Explain why is the residual so large.
vecnorm(A*U₁-U₁*diagm(λ))
```

Out[14]:
```
67.99890719481296
```

In [15]:
```
[λ sort(eigvals(A),rev=true)]
```

Out[15]:
```
20×2 Array{Float64,2}:
    6.22495e17      6.22495e17
    3.73973e16      3.73973e16
    4.18842e11      4.18842e11
    1.24604e6       1.24604e6
    1.90729e5       1.90729e5
 3078.7          3078.7
  943.337         943.337
  303.624         303.624
    0.0324829      23.8341
```

```
0.00290981      0.0324829
5.8684e-7       0.00290981
3.7354e-10      5.86835e-7
1.67425e-11     3.70478e-10
8.05676e-13     6.56641e-11
2.03386e-13     8.31129e-13
6.09339e-16     6.7895e-16
1.03927e-17     1.03439e-18
6.24441e-19    -4.55219e-13
1.83052e-19    -2.84635e-5
1.29513e-22    -1.65989
```

1.3.4 Example - Hilbert matrix

We need the newest version of the package SpecialMatrices.jl.

In [16]:
```
# Pkg.checkout("SpecialMatrices")
using SpecialMatrices
```

In [17]:
```
whos(SpecialMatrices)
```

```
             Cauchy     40 bytes  UnionAll
          Circulant     40 bytes  UnionAll
          Companion     40 bytes  UnionAll
          Frobenius     40 bytes  UnionAll
             Hankel     40 bytes  UnionAll
            Hilbert     40 bytes  UnionAll
              Kahan     80 bytes  UnionAll
            Riemann     40 bytes  UnionAll
    SpecialMatrices   6535 bytes  Module
             Strang     40 bytes  UnionAll
           Toeplitz     40 bytes  UnionAll
        Vandermonde     40 bytes  UnionAll
              embed      0 bytes  SpecialMatrices.#embed
```

In [18]:
```
C=Cauchy([1,2,3,4,5],[0,1,2,3,4])
```

Out[18]:
```
5×5 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 [19]:
```
H=Hilbert(5)
```

```
Out[19]: SpecialMatrices.Hilbert{Rational{Int64}}(5, 5)
```

```
In [20]: Hf=full(H)
```

```
Out[20]: 5×5 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 [21]: # This is exact
         det(Hf)
```

```
Out[21]: 1//266716800000
```

```
In [22]: # 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 all(isinteger,C.x)&all(isinteger,C.y)
                 return num//den
             else
                 return num/den
             end
         end
```

```
Out[22]: det (generic function with 23 methods)
```

```
In [23]: det(C)
```

```
Out[23]: 1//266716800000
```

We now compute componentwise highly accurate $A = LDL^T$ factorization of a Hilbert (Cauchy) matrix. Using `Rational` numbers gives high accuracy.

```
In [24]: # Exact LDLT factorization from Fact 8, no pivoting.
         function myLDLT(C::Cauchy)
             n=length(C.x)
             T=typeof(C.x[1])
             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[24]: myLDLT (generic function with 1 method)

In [25]:
```
L,D=myLDLT(C)
L
```

Out[25]:
```
5×5 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
```

In [26]:
```
D
```

Out[26]:
```
5-element Array{Rational{Int64},1}:
 1//1
 1//12
 1//180
 1//2800
 1//44100
```

In [27]:
```
L*diagm(D)*L' # -full(H)
```

Out[27]:
```
5×5 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 [28]:
```
# L*D*L' is an RRD
cond(L)
```

Out[28]: 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 [29]: 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[29]: myGECP (generic function with 1 method)
```

```
In [30]: # First a smaller test
         l=8
         C=Cauchy(collect(1:l),collect(0:l-1))
```

```
Out[30]: 8×8 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 [31]:
```
X,D,Y,pr,pc=myGECP(C)
```

Out[31]:
```
([1.0 0.0 ... 0.0 0.0; 0.333333 1.0 ... 0.0 0.0; ... ; 0.142857 0.714286 ... 1.0 0.0; 0
```

In [32]:
```
vecnorm((X*diagm(D)*Y')-full(C)[pr,pc])
```

Out[32]:
```
1.1527756336890508e-16
```

In [33]:
```
vecnorm(X[invperm(pr),:]*diagm(D)*Y[invperm(pc),:]'-full(C))
```

Out[33]:
```
1.1527756336890508e-16
```

In [34]:
```
# Now the big test.
n=100
H=Hilbert(n)
C=Cauchy(collect(1:n), collect(0:n-1))
```

Out[34]:
```
100×100 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 [35]: function myRRD(C::Cauchy)
             X,D,Y,pr,pc=myGECP(C)
             X[invperm(pr),:], D, Y[invperm(pc),:]
         end
```

```
Out[35]: myRRD (generic function with 1 method)
```

```
In [36]: X,D,Y=myRRD(C);
```

```
In [37]: # Check
         vecnorm((X*diagm(D)*Y')-full(C))
```

```
Out[37]: 3.053335078291665e-16
```

```
In [38]: # Is this RRD? here X=Y
         cond(X), cond(Y)
```

```
Out[38]: (72.24644120521842, 72.24644120521842)
```

```
In [39]: # 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[39]: myRRDSVD (generic function with 1 method)
```

```
In [40]: U,σ,V=myRRDSVD(X,D,Y);
```

```
In [41]: # Check residual and orthogonality
         vecnorm(full(C)*V-U*diagm(σ)), vecnorm(U'*U-I), vecnorm(V'*V-I)
```

```
Out[41]: (4.344717174348276e-5, 1.5138061246584667e-14, 2.815031354618813e-5)
```

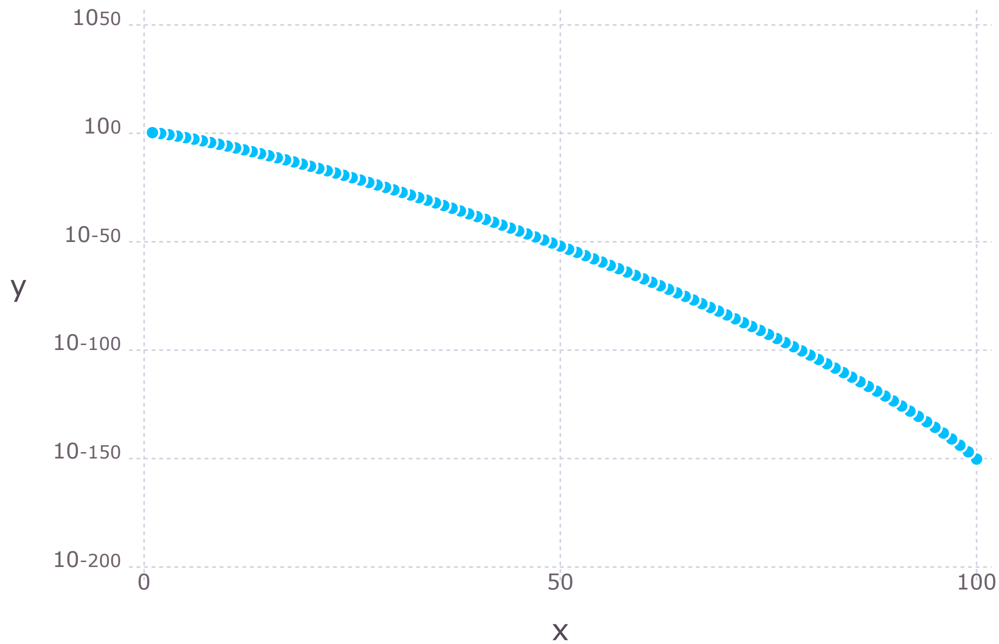
In [42]:
```
# Observe the difference!!
[sort(σ) sort(svdvals(C)) sort(eigvals(full(C)))]
```

Out[42]:
```
100×3 Array{Float64,2}:
 5.7797e-151    9.87732e-20  -4.33965e-16
 1.29735e-147   2.43276e-19  -3.88646e-16
 1.44439e-144   3.81207e-19  -3.2957e-16
 1.06342e-141   4.37274e-19  -2.36428e-16
 5.82434e-139   4.9596e-19   -1.52442e-16
 2.5311e-136    6.63107e-19  -1.22891e-16
 9.09071e-134   8.3494e-19   -1.0907e-16
 2.77536e-131   8.97432e-19  -8.46881e-17
 7.35195e-129   1.06175e-18  -6.79789e-17
 1.71656e-126   1.18149e-18  -6.2771e-17
 3.57648e-124   1.26493e-18  -5.90783e-17
 6.71629e-122   1.45426e-18  -5.19094e-17
 1.14619e-119   1.4791e-18   -4.34161e-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 [43]:
```
# Plot the eigenvalues (singular values) and left singular vectors
using Gadfly
```

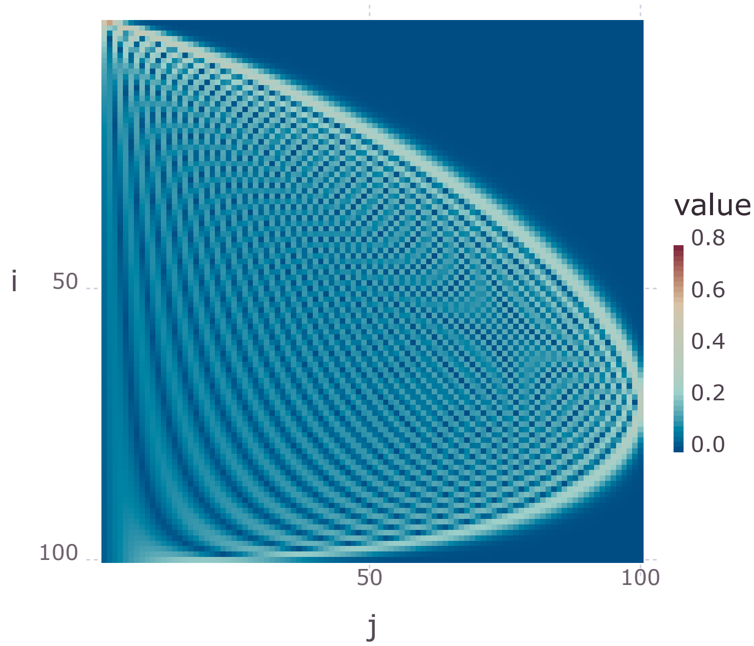
In [44]:
```
plot(x=collect(1:length(σ)),y=σ,Scale.y_log10)
```

Out[44]:




```
In [45]: spy(abs.(U))
```

Out[45]:



1.4 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.

1.4.1 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 $A = D + \rho z z^T$, 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 $i \neq j$.

1.4.2 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.

1.4.3 Example - Random arrowhead matrix

In [46]:
```
# Pkg.add("Arrowhead"); Pkg.checkout("Arrowhead")
using Arrowhead
```

In [47]:
```
whos(Arrowhead)
```

```
                    Arrowhead     43 KB     Module
                 GenHalfArrow      0 bytes  Arrowhead.#GenHalfArrow
                  GenSymArrow      0 bytes  Arrowhead.#GenSymArrow
                   GenSymDPR1      0 bytes  Arrowhead.#GenSymDPR1
                    HalfArrow     40 bytes  UnionAll
                     SymArrow     40 bytes  UnionAll
                      SymDPR1     40 bytes  UnionAll
                       bisect      0 bytes  Arrowhead.#bisect
                          eig      0 bytes  Base.LinAlg.#eig
                          inv      0 bytes  Base.#inv
                     rootsWDK      0 bytes  Arrowhead.#rootsWDK
                      rootsah      0 bytes  Arrowhead.#rootsah
                          svd      0 bytes  Base.LinAlg.#svd
                          tdc      0 bytes  Arrowhead.#tdc
```

In [48]:
```
methods(GenSymArrow)
```

Out[48]:
```
# 1 method for generic function "GenSymArrow":
GenSymArrow(n::Integer, i::Integer) in Arrowhead at C:\Users\Ivan\.julia\v0.6\Arrowhead
```

In [49]:
```
n=10
A=GenSymArrow(n,n)
```

Out[49]:
```
10×10 Arrowhead.SymArrow{Float64}:
 0.915498  0.0       0.0       0.0       ...  0.0       0.0       0.941099
 0.0       0.959933  0.0       0.0          0.0       0.0       0.643116
 0.0       0.0       0.603399  0.0          0.0       0.0       0.569521
 0.0       0.0       0.0       0.156622     0.0       0.0       0.414922
 0.0       0.0       0.0       0.0          0.0       0.0       0.00541137
 0.0       0.0       0.0       0.0       ...  0.0       0.0       0.95822
 0.0       0.0       0.0       0.0          0.0       0.0       0.153246
 0.0       0.0       0.0       0.0          0.436243  0.0       0.293209
 0.0       0.0       0.0       0.0          0.0       0.519064  0.57873
 0.941099  0.643116  0.569521  0.414922     0.293209  0.57873   0.924866
```

In [50]:
```
# Elements of the type SymArrow
A.D, A.z, A.a, A.i
```

Out[50]:
```
([0.915498, 0.959933, 0.603399, 0.156622, 0.0212465, 0.625239, 0.425763, 0.436243, 0.51
```

```
In [51]: tols=[1e2,1e2,1e2,1e2,1e2]
         U,λ=eig(A,tols)
         vecnorm(full(A)*U-U*diagm(λ)), vecnorm(U'*U-I)
```

```
Out[51]: (8.532217892124336e-16, 6.021857972601667e-16)
```

```
In [52]: # Timings - notice the O(n^2)
         @time eig(GenSymArrow(1000,1000),tols);
         @time eig(GenSymArrow(2000,2000),tols);
```

```
  0.788273 seconds (18.22 M allocations: 372.933 MiB, 26.56% gc time)
  2.515983 seconds (73.41 M allocations: 1.463 GiB, 5.15% gc time)
```

1.4.4 Example - Numerically demanding matrix

```
In [53]: 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, 1.0
```

```
Out[53]: 6×6 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 [54]: U,λ=eig(A,tols);
         println([sort(λ) sort(eigvals(full(A)))])
```

```
[-0.348142 -0.348142; 1.26185 1.26185; 2.22325 2.22325; 3.18832 3.18832; 4.17472 4.17472; 2.0e10
```

1.4.5 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

$$A^{-1}=D^{-1}+\gamma uu^T,\quad u=D^{-1}z,\quad \gamma=-\frac{\rho}{1+\rho z^T D^{-1}z},$$

   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 $w_1=-D_1^{-1}z_1\frac{1}{\zeta_i}$, $w_2=-D_2^{-1}z_2\frac{1}{\zeta_i}$, and $b=\frac{1}{\zeta_i^2}\left(\frac{1}{\rho}+z_1^T D_1^{-1}z_1+z_2^T D_2^{-1}z_2\right)$.

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.

1.4.6 Example - Random DPR1 matrix

In [55]:
```
n=10
A=GenSymDPR1(n)
```

Out[55]:
```
10×10 Arrowhead.SymDPR1{Float64}:
 1.27328     0.151103    0.0817164  …  0.511917    0.492817    0.181279
 0.151103    0.228829    0.01849        0.115831    0.111509    0.041018
 0.0817164   0.01849     0.249177       0.0626416   0.0603043   0.0221825
 0.567878    0.128494    0.0694893      0.43532     0.419077    0.154155
 0.29062     0.0657584   0.0355621      0.222781    0.214469    0.0788909
 0.254375    0.0575574   0.0311271  …  0.194997    0.187721    0.0690521
 0.53286     0.12057     0.0652043      0.408476    0.393235    0.144649
 0.511917    0.115831    0.0626416      1.25973     0.37778     0.138964
 0.492817    0.111509    0.0603043      0.37778     0.938297    0.133779
 0.181279    0.041018    0.0221825      0.138964    0.133779    0.755911
```

In [56]:
```
# Elements of the type SymDPR1
A.D, A.u, A.r
```

Out[56]: ([0.605481, 0.194639, 0.239178, 0.788892, 0.307731, 0.872574, 0.928589, 0.867306, 0.574

```
In [57]: U,λ=eig(A,tols)
         vecnorm(full(A)*U-U*diagm(λ)), vecnorm(U'*U-I)
```

Out[57]: (1.0540180993761921e-15, 6.156437145206418e-16)

1.4.7 Example - Numerically demanding matrix

```
In [58]: 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 ], 1
```

```
Out[58]: 6×6 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 [59]: U,λ=eig(A,tols)
         norm(full(A)*U-U*diagm(λ)), norm(U'*U-I), println([sort(λ) sort(eigvals(full(A)))])
```

```
[-5.0 -5.0; -0.004 -5.74215e-14; 1.0e-24 5.62712e-14; 0.004 0.004; 5.0 5.0; 1.0e20 1.0e20]
```

Out[59]: (3.0381820397056514e-6, 2.2204460858891437e-16, nothing)

In [ ]: