Lecture 6: Conditioning of linear systems, Matrix structure

Jamie Haddock

Table of contents

0.1 Conditioning of Linear Systems

We consider now the conditioning of solving the square linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$. Here, the data is $\mathbf{A}$ and $\mathbf{b}$, and the solution is $\mathbf{x}$.

For simplicity, we'll imagine that there are perturbations only to $\mathbf{b}$, while $\mathbf{A}$ is fixed. Suppose $\mathbf{A}\mathbf{x} = \mathbf{b}$ is perturbed to

$$\mathbf{A}(\mathbf{x}+\mathbf{h}) = \mathbf{b}+\mathbf{d}.$$

The condition number is the relative change in the solution divided by the relative change in the data,

$$\frac{\frac{\|\mathbf{h}\|}{\|\mathbf{x}\|}}{\frac{\|\mathbf{d}\|}{\|\mathbf{b}\|}} = \frac{\|\mathbf{h}\|\|\mathbf{b}\|}{\|\mathbf{d}\|\|\mathbf{x}\|}.$$

Since $\mathbf{h} = \mathbf{A}^{-1}\mathbf{d}$, we can bound $\|\mathbf{h}\|$ as

$$\|\mathbf{h}\| \le \|\mathbf{A}^{-1}\|\|\mathbf{d}\|.$$

Similarly, we have $\|\mathbf{b}\| \le \|\mathbf{A}\|\|\mathbf{x}$ and so

$$\frac{\|\mathbf{h}\|\|\mathbf{b}\|}{\|\mathbf{d}\|\|\mathbf{x}\|} \le \frac{\|\mathbf{A}^{-1}\|\|\mathbf{d}\|\|\mathbf{A}\|\|\mathbf{x}\|}{\|\mathbf{d}\|\|\mathbf{x}\|} = \|\mathbf{A}^{-1}\|\|\mathbf{A}\|.$$

This bound is tight – the inequalities are equations for some choices of $\mathbf{b}$ and $\mathbf{d}$.

Definition: Matrix condition number

The **matrix condition number** of an invertible square matrix $\mathbf{A}$ is

$$\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\|\|\mathbf{A}\|.$$

This value depends on the choice of norm; a subscript on κ such as 1, 2, or ∞ is used if clarification is needed. If $\mathbf{A}$ is singular, we define $\kappa(\mathbf{A}) = \infty$.

Theorem: Conditioning of linear systems

If $\mathbf{A}(\mathbf{x}+\triangle\mathbf{x}) = \mathbf{b}+\triangle\mathbf{b}$, then

$$\frac{\|\triangle\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A})\frac{\|\triangle\mathbf{b}\|}{\|\mathbf{b}\|}.$$

If $(\mathbf{A}+\triangle\mathbf{A})(\mathbf{x}+\triangle\mathbf{x}) = \mathbf{b}$, then

$$\frac{\|\triangle\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A})\frac{\|\triangle\mathbf{A}\|}{\|\mathbf{A}\|},$$

in the limit $\|\triangle\mathbf{A}\| \to 0$.

Exercise: Lower bound on condition number

Show that $\kappa(\mathbf{A}) \ge 1$.

Answer:

We have $1 = \|\mathbf{I}\| = \|\mathbf{A}\mathbf{A}^{-1}\| \le \|\mathbf{A}\|\|\mathbf{A}^{-1}\| = \kappa(\mathbf{A})$.

A condition number equal to 1 is the best we can hope for – this means the relative perturbation in the solution is the same size as that of the data. If a matrix has condition number 10^t indicates that in floating-point arithmetic, roughly t digits are lost in computing the solution $\mathbf{x}$. If $\kappa(\mathbf{A}) > 1/\epsilon_{\text{mach}}$, then for numerical purposes, the matrix $\mathbf{A}$ is effectively singular.

Julia has the function `cond` to compute the matrix condition number. The ℓ_2 norm is used by default in this calculation. We'll begin with an example of a *Hilbert matrix* which is famously ill-conditioned.

```
using LinearAlgebra

A = [ 1/(i+j) for i in 1:6, j in 1:6 ]
 = cond(A)
```

```
5.109816297946132e7
```

When solving a linear system with this matrix, we will lose nearly 8 digits of accuracy due to the ill-conditioning of this problem!

```
x = 1:6
b = A*x;
```

We perturb the system randomly by 10^{-10} in norm.

```
 A = randn(size(A));  A = 1e-10*( A/opnorm( A));
 b = randn(size(b));  b = 1e-10*normalize( b);
```

We solve the perturbed problem and see how the solution is changled.

```
new_x = ((A +  A) \ (b+ b))
 x = new_x - x
```

```
6-element Vector{Float64}:
 -7.449594121577974e-6
  0.0001247466230993588
 -0.0006403322152883639
```

```
  0.0013944543468378257
 -0.0013561726908424276
  0.00048554115369814355
```

```
@show relative_error = norm( x) / norm(x);
```

```
relative_error = norm( x) / norm(x) = 0.00022101414770238344
```

```
println("Upper bound due to b: $( *norm( b)/norm(b))")
println("Upper bound due to A: $( *norm( A)/norm(A))")
```

```
Upper bound due to b: 0.0006723667714371329
Upper bound due to A: 0.007039260527116223
```

These errors are due to our manual perturbations we made to the data. Even just machine roundoff perturbs this data and affects the solution of this ill-conditioned problem. This error will scale with ϵ_{mach}.

```
 x = A\b - x
@show relative_error = norm( x)/norm(x);
@show rounding_bound =  *eps();
```

```
relative_error = norm( x) / norm(x) = 7.822650774976615e-10
rounding_bound =   * eps() = 1.134607141116935e-8
```

Larger Hilbert matrices are even more ill-conditioned and their linear systems suffer from more error during solution.

```
A = [ 1/(i+j) for i=1:14, j=1:14 ];
  = cond(A)                               #exceeds 1/eps()
```

```
5.802584125151949e17
```

```
rounding_bound =  *eps()
```

```
128.8432499613623
```

```
x = 1:14
b = A*x
 x = A\b - x
@show relative_error = norm( x)/norm(x);
```

```
relative_error = norm( x) / norm(x) = 4.469466154206132
```

There are zero accurate digits!

0.2 Residual and backward error

When we don't know the solution of a linear system, we cannot compare our approximate computed solution to the true solution, so we use the residual error.

Definition: Residual of a linear system

For the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$, the **residual** at a solution estimate $\hat{\mathbf{x}}$ is

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}.$$

A zero residual means we have an exact solution, and if the matrix is rank n, then we have $\hat{\mathbf{x}} = \mathbf{x}$.

More generally, though, we have

$$\mathbf{A}\hat{\mathbf{x}} = \mathbf{b} - \mathbf{r}.$$

This means that $\hat{\mathbf{x}}$ is an exact solution for a linear system with right hand error changed by $-\mathbf{r}$.

This is what we search for when studying background error!

Hence, residual error of a linear system is the system's backward error. We can connect this error to the forward error by making the definition $\mathbf{h} = \hat{\mathbf{x}} - \mathbf{x}$ in the equation $\mathbf{A}(\mathbf{x} + \mathbf{h}) = \mathbf{b} + \mathbf{d}$.

Then

$$\mathbf{d} = \mathbf{A}(\mathbf{x} + \mathbf{h}) - \mathbf{b} = \mathbf{A}\mathbf{h} = -\mathbf{r}.$$

Thus, our previous theorem yields

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A}) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}.$$

The relationship between relative error and the relative residual is scaling by the matrix condition number.

Fact:

When solving a linear system, we can only expect that the backward (residual) error is small, not the error, since this will suffer from scaling by the matrix condition number.

1 Matrix structure

Many matrices typically encountered in scientific computing have special structure. It can be *very* helpful to understand and exploit these special structures!

1.1 Diagonal dominance

An $n \times n$ matrix $\mathbf{A}$ is **(row) diagonally dominant** if

$$|A_{ii}| > \sum_{j=1//j\neq i}^{n} |A_{ij}| \text{ for each } i = 1, \cdots, n.$$

This says that the magnitude of entries on the diagonal are larger than the sum of magnitudes of entries in the same row off-diagonal.

- Diagonally dominant matrices are guaranteed to be invertible.
- Diagonally dominant matrices do not need row-pivoting for elimination/LU stability.

1.2 Banded matrices

Definition: Bandwidth

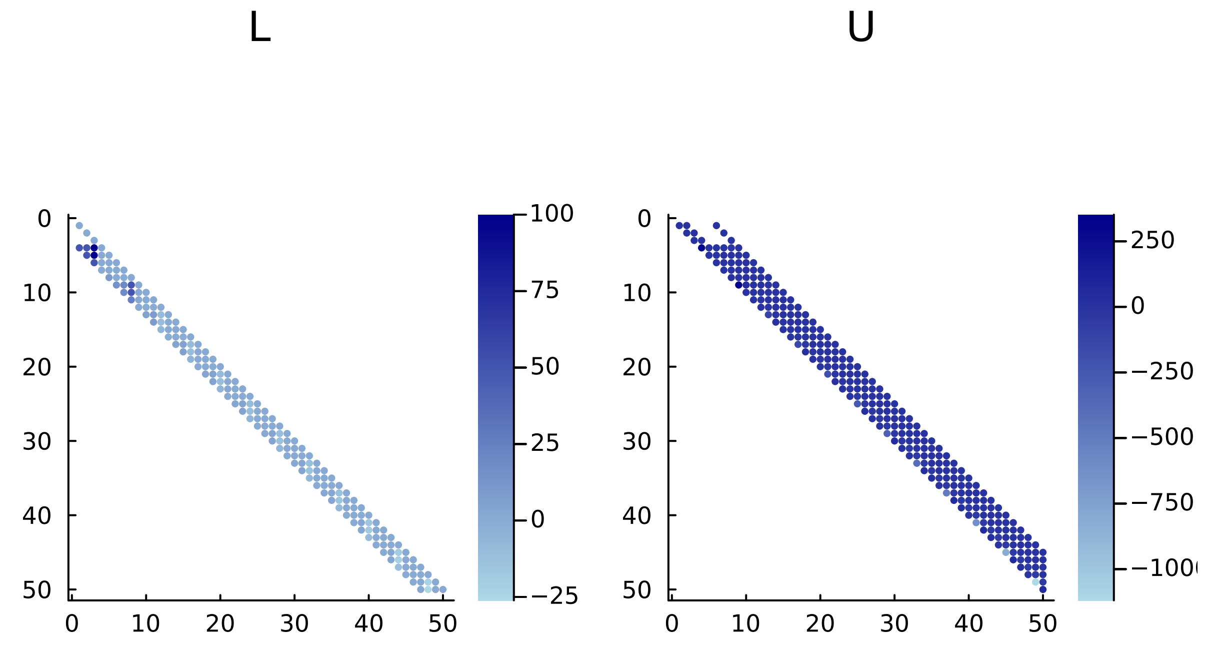
A matrix $\mathbf{A}$ has **upper bandwidth** b_u if $j - i > b_u$ implies $A_{ij} = 0$, and **lower bandwidth** b_l if $i - j > b_l$ implies $A_{ij} = 0$. We say the total **bandwidth** is $b_u + b_l + 1$. When $b_u = b_l = 1$, we have the important case of a **tridiagonal matrix**.

```
using SparseArrays

n = 50;
A = spdiagm( -3=>fill(n,n-3),
                0=>ones(n),
                1=>-(1:n-1),
                5=>fill(0.1,n-5) )
Matrix(A[1:7,1:7])
```

```
7×7 Matrix{Float64}:
  1.0  -1.0   0.0   0.0   0.0   0.1   0.0
  0.0   1.0  -2.0   0.0   0.0   0.0   0.1
  0.0   0.0   1.0  -3.0   0.0   0.0   0.0
 50.0   0.0   0.0   1.0  -4.0   0.0   0.0
  0.0  50.0   0.0   0.0   1.0  -5.0   0.0
  0.0   0.0  50.0   0.0   0.0   1.0  -6.0
  0.0   0.0   0.0  50.0   0.0   0.0   1.0
```

```
using FundamentalsNumericalComputation

L,U = FNC.lufact(A);
```

```
plot(layout=2)
spy!(sparse(L),m=2,subplot=1,title="L",color=:blues)
spy!(sparse(U),m=2,subplot=2,title="U",color=:blues)
```



The LU factors are also banded!

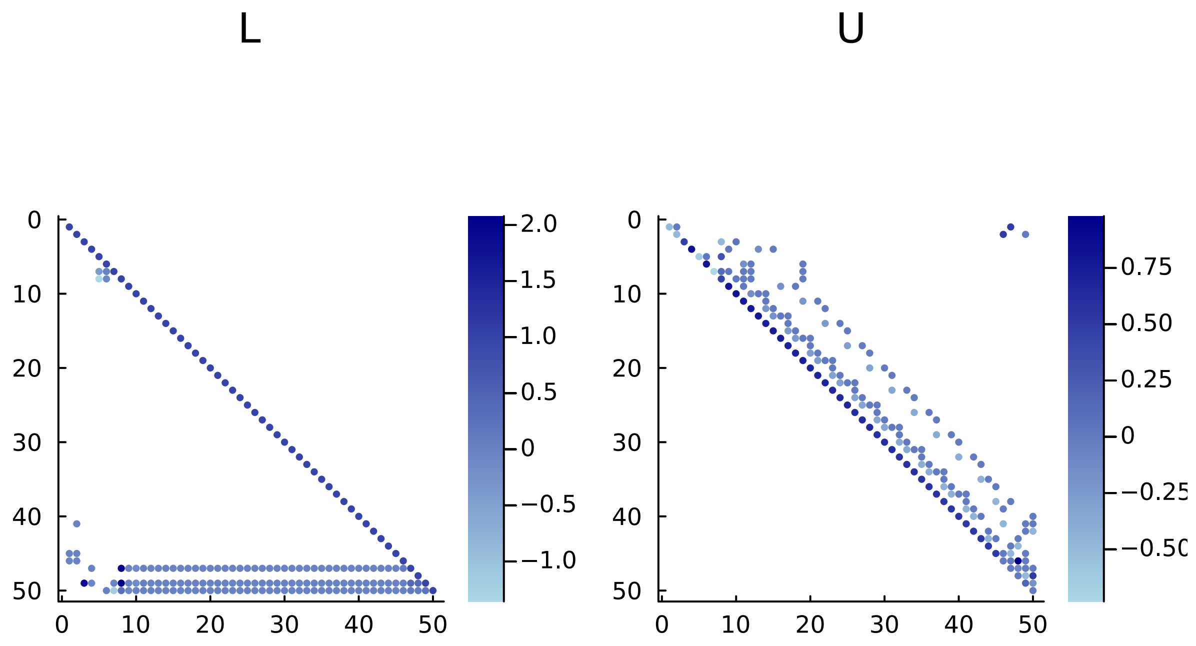
Note:

The number of flops needed by LU factorization without pivoting is $\mathcal{O}(b_u b_t n)$ when the upper and lower bandwidths are b_u and b_l.

However, using row pivoting actually can expand or destroy bandedness!

```
fact = lu(A);
```

```
plot(layout=2)
spy!(sparse(fact.L),m=2,subplot=1,title="L",color=:blues)
spy!(sparse(fact.U),m=2,subplot=2,title="U",color=:blues)
```



In order for Julia to take advantage of banded matrix advantages if we use an ordinary (dense) matrix representation (since it doesn't know in advance where the zeros are).

```
n = 10000
A = diagm(0=>1:n, 1=>n-1:-1:1, -1=>ones(n-1))
lu(rand(3,3)) #throwaway to force compilation
@time lu(A);
```

```
  3.492128 seconds (7 allocations: 763.016 MiB, 0.19% gc time)
```

If we use a sparse matrix representation, the speedup is dramatic!

```
A = spdiagm(0=>1:n, 1=>n-1:-1:1, -1=>ones(n-1))
lu(A); #throwaway for sparse compile
@time lu(A);
```

```
  0.004157 seconds (86 allocations: 9.920 MiB)
```

1.3 Sparse matrices

Extremely large matrices cannot be stored in primary memory of a computer unless they are **sparse** – that is, they have few nonzero entries. A sparse matrix has *structural zeros*, entries that are known to be zero and thus no value need be stored.

Sparse matrices are not (should not be) represented as a usual matrix array in memory. Instead, one can use one of a variety of sparse matrix representations.

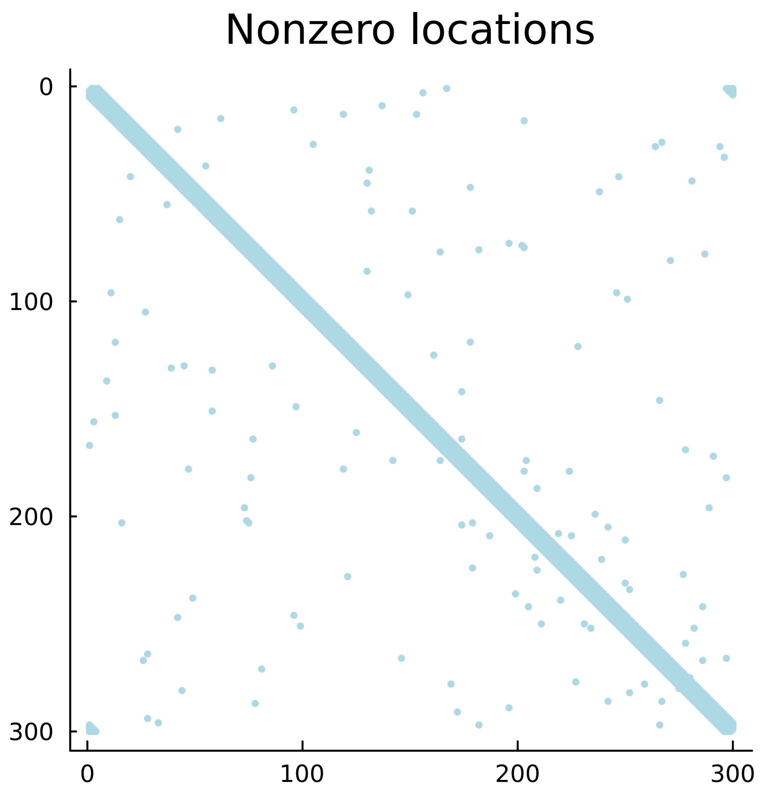
For example, you can store triples (i, j, A_{ij}) for all locations of nonzeros (i, j) in the matrix. This requires $3\text{nnz}(A)$ storage, whereas usual storage requires $\mathcal{O}(n^2)$ storage – this can be a *significant* advantage when $\text{nnz}(A) \ll n^2$.

A common source of sparse matrices is graphs or networks – large graphs often have few edges and thus their adjacency matrices (and other matrix representations) are often large, very sparse matrices!

```julia
using Graphs

G = Graphs.SimpleGraphs.newman_watts_strogatz(300,8,0.05)
A = Graphs.LinAlg.adjacency_matrix(G)
graphplot(A,linealpha=0.5)
```

```
LoadError: ArgumentError: Package Graphs not found in current path.
- Run `import Pkg; Pkg.add("Graphs")` to install the Graphs package.
ArgumentError: Package Graphs not found in current path.
- Run `import Pkg; Pkg.add("Graphs")` to install the Graphs package.

Stacktrace:
 [1] macro expansion
   @ ./loading.jl:2296 [inlined]
 [2] macro expansion
   @ ./lock.jl:273 [inlined]
 [3] __require(into::Module, mod::Symbol)
   @ Base ./loading.jl:2271
 [4] #invoke_in_world#3
   @ ./essentials.jl:1089 [inlined]
 [5] invoke_in_world
   @ ./essentials.jl:1086 [inlined]
 [6] require(into::Module, mod::Symbol)
   @ Base ./loading.jl:2260
```

```julia
spy(A,title="Nonzero locations", m=2, color=:blues)
```



```
m,n = size(A)
@show density = nnz(A) / (m*n);
```

```
density = nnz(A) / (m * n) = 0.028066666666666667
```

This is actually a relatively dense graph. Many real-world network datasets are *far* more sparse!

The computer memory consumed by any variable can be learned by using the **summarysize** command. We see the storage savings offered by sparse matrix representations is dramatic!

```
F = Matrix(A) #this is the dense matrix representation of A
Base.summarysize(F)/Base.summarysize(A)
```

```
16.751535455053045
```

Matrix-vector products are also much more efficient when the matrix is given in sparse form, because the operations using structural zeros are completely skipped.

```
x = randn(n)
b = A*x; #throwaway for load/compilation of *
@elapsed for i in 1:300; A*x; end #run 300 times to get a good estimation of average time required
```

```
0.000903223
```

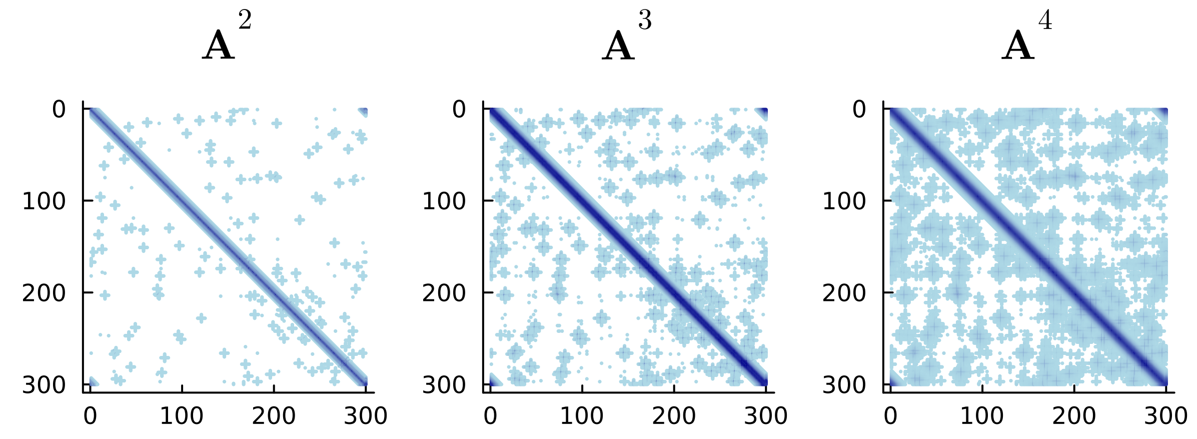
```
F*x;
@elapsed for i in 1:300; F*x; end
```

```
0.010426952
```

Computer arithmetic operations exploit sparsity whenever they can and calculations on sparse matrices can be much more efficient than calculations on their dense counterparts!

However, some operations are not guaranteed to preserve sparsity (mathematically) – this phenomenon is known as **fill-in**.

```
plt = plot(layout=(1,3),legend=:none,size=(600,240))
for k in 2:4
    spy!(A^k,subplot=k-1,color=:blues,title=latexstring("\\mathbf{A}^$k"))
end
plt
```



1.4 Symmetric matrices

The LU decomposition for a symmetric matrix (if it exists), takes on a special form:

$$\mathbf{A} = \mathbf{LDL}^\top.$$

Note:

$\mathbf{LDL}^\top$ factorization on an $n \times n$ symmetric matrix, when successful, takes $\sim \frac{1}{3}n^3$ flops – half as many as is necessary for regular **LU** factorization.

1.5 Symmetric positive definite matrices

Suppose $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{x} \in \mathbb{R}^n$. Note that $\mathbf{x}^\top\mathbf{A}\mathbf{x}$ is scalar valued – this is called a **quadratic form**.

Definition: Symmetric positive (semi-)definite matrix

A real $n \times n$ matrix $\mathbf{A}$ is called a **symmetric positive definite matrix** (SPD) if it is symmetric and, for all $\mathbf{x} \neq 0$,

$$\mathbf{x}^\top\mathbf{A}\mathbf{x} > 0.$$

A matrix is a **symmetric positive definite matrix** if the inequality above holds but possibly with equality for some nonzero vectors $\mathbf{x}$.

This definition, in combination with knowledge of either the spectral decomposition or $\mathbf{LDL}^\top$ factorization, allows one to prove the following theorem.

Theorem: Cholesky factorization

Any SPD matrix $\mathbf{A}$ may be factored as

$$\mathbf{A} = \mathbf{R}^\top \mathbf{R},$$

where $\mathbf{R}$ is an upper triangular matrix with positive diagonal elements. This is called the **Cholesky factorization**.

Note:

Cholesky factorization of an $n \times n$ SPD matrix takes $\sim \frac{1}{3}n^3$ flops.

```
A = rand(1.0:9.0,4,4)
B = A + A'              #easy symmetrization technique!
cholesky(B)
```

```
LoadError: PosDefException: matrix is not positive definite; Factorization failed.
PosDefException: matrix is not positive definite; Factorization failed.

Stacktrace:
  [1] checkpositivedefinite
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [2] #cholesky!#163
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [3] cholesky!
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [4] #cholesky!#164
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [5] cholesky! (repeats 2 times)
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [6] _cholesky
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [7] cholesky(A::Matrix{Float64}, ::NoPivot; check::Bool)
    @ LinearAlgebra ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/Lin
  [8] cholesky
    @ ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/LinearAlgebra/src
  [9] cholesky(A::Matrix{Float64})
    @ LinearAlgebra ~/.julia/juliaup/julia-1.11.2+0.aarch64.apple.darwin14/share/julia/stdlib/v1.11/Lin
 [10] top-level scope
    @ In[25]:3
```

```
#must be careful to build an SPD matrix2graph
B = A'*A
cf = cholesky(B)
```

```
Cholesky{Float64, Matrix{Float64}}
U factor:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 9.94987  10.0504  10.5529    8.44232
           9.4863   7.68892   5.91922
                    3.93915  -1.42247
                              5.16398
```

```
R = cf.U

opnorm(R'*R - B) / opnorm(B)  #relative error in factorization is near 0!
```

```
2.6382068635067203e-17
```