Table of Contents

In [1]:

```
versioninfo()
```

```
Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: macOS (x86_64-apple-darwin14.5.0)
  CPU: Intel(R) Core(TM) i7-6920HQ CPU @ 2.90GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = code
```

Cholesky Decomposition

- A basic tenet in numerical analysis:

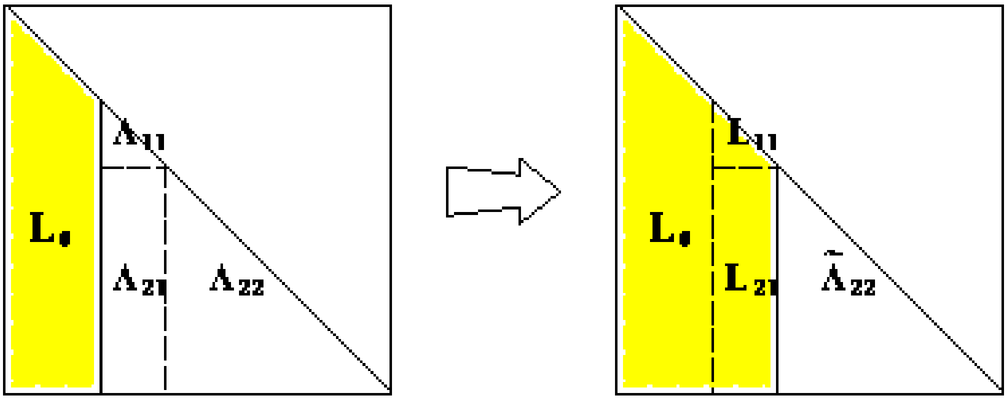
> **The structure should be exploited whenever solving a problem.**

Common structures include: symmetry, positive (semi)definiteness, sparsity, Kronecker product, low rank, ...

- LU decomposition (Gaussian Elimination) is **not** used in statistics so often because most of time statisticians deal with positive (semi)definite matrix. (That's why I hate to see `solve()` in R code.)
- For example, in the normal equation
$$\mathbf{X}^T \mathbf{X} \beta = \mathbf{X}^T \mathbf{y}$$
for linear regression, the coefficient matrix $\mathbf{X}^T \mathbf{X}$ is symmetric and positive semidefinite. How to exploit this structure?

Cholesky decomposition

- **Theorem**: Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be symmetric and positive definite. Then $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is lower triangular with positive diagonal entries and is unique.
  Proof (by induction):
  If $n = 1$, then $\ell = \sqrt{a}$. For $n > 1$, the block equation
  $$\begin{pmatrix} a_{11} & \mathbf{a}^T \\ \mathbf{a} & \mathbf{A}_{22} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \mathbf{0}_{n-1}^T \\ \mathbf{l} & \mathbf{L}_{22} \end{pmatrix} \begin{pmatrix} \ell_{11} & \mathbf{l}^T \\ \mathbf{0}_{n-1} & \mathbf{L}_{22}^T \end{pmatrix}$$
  has solution
  $$\begin{aligned} \ell_{11} &= \sqrt{a_{11}} \\ \mathbf{l} &= \ell_{11}^{-1}\mathbf{a} \\ \mathbf{L}_{22}\mathbf{L}_{22}^T &= \mathbf{A}_{22} - \mathbf{l}\mathbf{l}^T = \mathbf{A}_{22} - a_{11}^{-1}\mathbf{a}\mathbf{a}^T. \end{aligned}$$
  Now $a_{11} > 0$ (why?), so ℓ_{11} and $\mathbf{l}$ are uniquely determined. $\mathbf{A}_{22} - a_{11}^{-1}\mathbf{a}\mathbf{a}^T$ is positive definite because $\mathbf{A}$ is positive definite (why?). By induction hypothesis, $\mathbf{L}_{22}$ exists and is unique.
- The constructive proof completely specifies the algorithm:



- Computational cost:
  $$\frac{1}{2}[2(n-1)^2 + 2(n-2)^2 + \cdots + 2 \cdot 1^2] \approx \frac{1}{3}n^3 \quad \text{flops}$$
  plus n square roots. Half the cost of LU decomposition by utilizing symmetry.
- In general Cholesky decomposition is very stable. Failure of the decomposition simply means $\mathbf{A}$ is not positive definite. It is an efficient way to test positive definiteness.

Pivoting

- When $\mathbf{A}$ does not have full rank, e.g., $\mathbf{X}^T\mathbf{X}$ with a non-full column rank $\mathbf{X}$, we encounter $a_{kk} = 0$ during the procedure.
- **Symmetric pivoting**. At each stage k, we permute both row and column such that $\max_{k \le i \le n} a_{ii}$ becomes the pivot. If we encounter $\max_{k \le i \le n} a_{ii} = 0$, then $\mathbf{A}[k:n, k:n] = \mathbf{0}$ (why?) and the algorithm terminates.
- With symmetric pivoting:
  $$\mathbf{P}\mathbf{A}\mathbf{P}^T = \mathbf{L}\mathbf{L}^T,$$
  where $\mathbf{P}$ is a permutation matrix and $\mathbf{L} \in \mathbb{R}^{n \times r}$, $r = \text{rank}(\mathbf{A})$.

Implementation

- LAPACK functions: [`?potrf`](http://www.netlib.org/lapack/explore-html/d1/d7a/group__double_p_ocomputational_ga2f55f604a6003d03b5cd4a0adcfb74d6.html#ga2f55f604) (without pivoting), [`?pstrf`](http://www.netlib.org/lapack/explore-html/da/dba/group__double_o_t_h_e_rcomputational_ga31cdc13a7f4ad687f4aefebff870e1cc.html#ga31c) (with pivoting).
- Julia functions: [`cholesky`](https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.cholesky), [`cholesky!`](https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.cholesky!), or call LAPACK wrapper functions [`potrf!`](https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.LAPACK.potrf!) and [`pstrf!`](https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.LAPACK.pstrf!)

Example: positive definite matrix.

In [2]:

```julia
using LinearAlgebra

A = Float64.([4 12 -16; 12 37 -43; -16 -43 98])
```

Out[2]:

```
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0
```

In [3]:

```julia
# Cholesky without pivoting
Achol = cholesky(A)
```

Out[3]:

```
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0
```

In [4]:

```julia
typeof(Achol)
```

Out[4]:

```
Cholesky{Float64,Array{Float64,2}}
```

In [5]:

```julia
fieldnames(typeof(Achol))
```

Out[5]:

```
(:factors, :uplo, :info)
```

In [6]:

```
# retrieve the lower triangular Cholesky factor
Achol.L
```

Out[6]:

```
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ⋅    ⋅ 
  6.0  1.0   ⋅ 
 -8.0  5.0  3.0
```

In [7]:

```
# retrieve the upper triangular Cholesky factor
Achol.U
```

Out[7]:

```
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0
```

In [8]:

```
b = [1.0; 2.0; 3.0]
A \ b # this does LU; wasteful!; 2/3 n^3 + 2n^2
```

Out[8]:

```
3-element Array{Float64,1}:
 28.58333333333338
 -7.666666666666679
  1.3333333333333353
```

In [9]:

```
Achol \ b # two triangular solves; only 2n^2 flops
```

Out[9]:

```
3-element Array{Float64,1}:
 28.583333333333332
 -7.666666666666666
  1.3333333333333333
```

In [10]:

```
det(A) # this actually does LU; wasteful!
```

Out[10]:

```
35.99999999999994
```

In [11]:

```
det(Achol) # cheap
```

Out[11]:

```
36.0
```

In [12]:

```
inv(A) # this does LU!
```

Out[12]:

```
3×3 Array{Float64,2}:
  49.3611   -13.5556     2.11111
 -13.5556     3.77778   -0.555556
   2.11111   -0.555556   0.111111
```

In [13]:

```
inv(Achol)
```

Out[13]:

```
3×3 Array{Float64,2}:
  49.3611   -13.5556     2.11111
 -13.5556     3.77778   -0.555556
   2.11111   -0.555556   0.111111
```

Example: positive semi-definite matrix.

In [14]:

```
using Random

Random.seed!(123) # seed
A = randn(5, 3)
A = A * transpose(A) # A has rank 3
```

Out[14]:

```
5×5 Array{Float64,2}:
  1.97375    2.0722    1.71191    0.253774   -0.544089
  2.0722     5.86947   3.01646    0.93344    -1.50292
  1.71191    3.01646   2.10156    0.21341    -0.965213
  0.253774   0.93344   0.21341    0.393107   -0.0415803
 -0.544089  -1.50292  -0.965213  -0.0415803   0.546021
```

In [15]:

```
Achol = cholesky(A, Val(true)) # 2nd argument requests partial pivoting
```

```
RankDeficientException(1)

Stacktrace:
 [1] chkfullrank at /Users/osx/buildbot/slave/package_osx64/build/usr/shar
e/julia/stdlib/v1.1/LinearAlgebra/src/cholesky.jl:498 [inlined]
 [2] #cholesky!#98(::Float64, ::Bool, ::Function, ::Hermitian{Float64,Arra
y{Float64,2}}, ::Val{true}) at /Users/osx/buildbot/slave/package_osx64/bui
ld/usr/share/julia/stdlib/v1.1/LinearAlgebra/src/cholesky.jl:195
 [3] #cholesky! at ./none:0 [inlined]
 [4] #cholesky!#100(::Float64, ::Bool, ::Function, ::Array{Float64,2}, ::V
al{true}) at /Users/osx/buildbot/slave/package_osx64/build/usr/share/juli
a/stdlib/v1.1/LinearAlgebra/src/cholesky.jl:221
 [5] #cholesky#102 at ./none:0 [inlined]
 [6] cholesky(::Array{Float64,2}, ::Val{true}) at /Users/osx/buildbot/slav
e/package_osx64/build/usr/share/julia/stdlib/v1.1/LinearAlgebra/src/choles
ky.jl:296
 [7] top-level scope at In[15]:1
```

In [16]:

```
Achol = cholesky(A, Val(true), check=false) # turn off checking pd
```

Out[16]:

```
CholeskyPivoted{Float64,Array{Float64,2}}
U factor with rank 4:
5×5 UpperTriangular{Float64,Array{Float64,2}}:
 2.4227  0.855329  0.38529    -0.620349    1.24508
  ⋅      1.11452  -0.0679895  -0.0121011   0.580476
  ⋅       ⋅        0.489935    0.4013     -0.463002
  ⋅       ⋅         ⋅          1.49012e-8  0.0
  ⋅       ⋅         ⋅           ⋅          0.0
permutation:
5-element Array{Int64,1}:
 2
 1
 4
 5
 3
```

In [17]:

```
rank(Achol) # determine rank from Cholesky factor
```

Out[17]:

```
4
```

In [18]:

```
rank(A) # determine rank from SVD, which is more numerically stable
```

Out[18]:

```
3
```

In [19]:

```
Achol.L
```

Out[19]:

```
5×5 LowerTriangular{Float64,Array{Float64,2}}:
  2.4227      ⋅           ⋅          ⋅            ⋅ 
  0.855329   1.11452      ⋅          ⋅            ⋅ 
  0.38529   -0.0679895   0.489935   ⋅            ⋅ 
 -0.620349  -0.0121011   0.4013     1.49012e-8   ⋅ 
  1.24508    0.580476   -0.463002  0.0          0.0
```

In [20]:

```
Achol.U
```

Out[20]:

```
5×5 UpperTriangular{Float64,Array{Float64,2}}:
 2.4227  0.855329   0.38529    -0.620349     1.24508 
  ⋅      1.11452   -0.0679895  -0.0121011    0.580476
  ⋅       ⋅         0.489935    0.4013      -0.463002
  ⋅       ⋅          ⋅          1.49012e-8   0.0     
  ⋅       ⋅          ⋅           ⋅          0.0     
```

In [21]:

```
Achol.p
```

Out[21]:

```
5-element Array{Int64,1}:
 2
 1
 4
 5
 3
```

In [22]:

```
# P A P' = L U
norm(Achol.P * A * Achol.P - Achol.L * Achol.U)
```

Out[22]:

```
7.528590393469295
```

Applications

- **No inversion** mentality: Whenever we see matrix inverse, we should think in terms of solving linear equations. If the matrix is positive (semi)definite, use Cholesky decomposition, which is twice cheaper than LU decomposition.

Multivariate normal density

Multivariate normal density $\text{MVN}(0, \Sigma)$, where Σ is p.d., is

$$-\frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det\Sigma - \frac{1}{2}\mathbf{y}^T\Sigma^{-1}\mathbf{y}.$$

- Method 1: (a) compute explicit inverse Σ^{-1} ($2n^3$ flops), (b) compute quadratic form ($2n^2 + 2n$ flops), (c) compute determinant ($2n^3/3$ flops).
- Method 2: (a) Cholesky decomposition $\Sigma = \mathbf{L}\mathbf{L}^T$ ($n^3/3$ flops), (b) Solve $\mathbf{L}\mathbf{x} = \mathbf{y}$ by forward substitutions (n^2 flops), (c) compute quadratic form $\mathbf{x}^T\mathbf{x}$ ($2n$ flops), and (d) compute determinant from Cholesky factor (n flops).

Which method is better?

In [23]:

```
# this is a person w/o numerical analsyis training
function logpdf_mvn_1(y::Vector, Σ::Matrix)
    n = length(y)
    - (n//2) * log(2π) - (1//2) * logdet(Σ) - (1//2) * y' * inv(Σ) * y
end

# this is an efficiency-savvy person
function logpdf_mvn_2(y::Vector, Σ::Matrix)
    n = length(y)
    Σchol = cholesky(Symmetric(Σ))
    - (n//2) * log(2π) - (1//2) * logdet(Σchol) - (1//2) * sum(abs2, Σchol.L \ y)
end

# better memory efficiency
function logpdf_mvn_3(y::Vector, Σ::Matrix)
    n = length(y)
    Σchol = cholesky(Symmetric(Σ))
    - (n//2) * log(2π) - (1//2) * logdet(Σchol) - (1//2) * dot(y, Σchol \ y)
end
```

Out[23]:

```
logpdf_mvn_3 (generic function with 1 method)
```

In [24]:

```
using BenchmarkTools, Distributions, Random

Random.seed!(123) # seed

n = 1000
# a pd matrix
Σ = convert(Matrix{Float64}, Symmetric([i * (n - j + 1) for i in 1:n, j in 1:n]))
y = rand(MvNormal(Σ)) # one random sample from N(0, Σ)

# at least they give same answer
@show logpdf_mvn_1(y, Σ)
@show logpdf_mvn_2(y, Σ)
@show logpdf_mvn_3(y, Σ);
```

```
logpdf_mvn_1(y, Σ) = -4878.375103770505
logpdf_mvn_2(y, Σ) = -4878.375103770553
logpdf_mvn_3(y, Σ) = -4878.375103770553
```

In [25]:

```
@benchmark logpdf_mvn_1(y, Σ)
```

Out[25]:

```
BenchmarkTools.Trial:
  memory estimate:  15.78 MiB
  allocs estimate:  14
  --------------
  minimum time:     37.747 ms (0.00% GC)
  median time:      41.845 ms (3.56% GC)
  mean time:        42.982 ms (3.71% GC)
  maximum time:     85.951 ms (54.02% GC)
  --------------
  samples:          117
  evals/sample:     1
```

In [26]:

```
@benchmark logpdf_mvn_2(y, Σ)
```

Out[26]:

```
BenchmarkTools.Trial:
  memory estimate:  15.27 MiB
  allocs estimate:  10
  --------------
  minimum time:     7.946 ms (0.00% GC)
  median time:      9.517 ms (16.00% GC)
  mean time:        9.518 ms (12.95% GC)
  maximum time:     59.474 ms (82.87% GC)
  --------------
  samples:          525
  evals/sample:     1
```

In [27]:

```
@benchmark logpdf_mvn_3(y, Σ)
```

Out[27]:

```
BenchmarkTools.Trial:
  memory estimate:  7.64 MiB
  allocs estimate:  8
  --------------
  minimum time:     6.353 ms (0.00% GC)
  median time:      6.571 ms (0.00% GC)
  mean time:        7.318 ms (9.51% GC)
  maximum time:     54.287 ms (88.08% GC)
  --------------
  samples:          682
  evals/sample:     1
```

- To evaluate same multivariate normal density at many observations $y_1, y_2, \ldots$, we pre-compute the Cholesky decomposition ($n^3/3$ flops), then each evaluation costs n^2 flops.

Linear regression

- Cholesky decomposition is **one** approach to solve linear regression. Assume $\mathbf{X} \in \mathbb{R}^{n \times p}$ and $\mathbf{y} \in \mathbb{R}^n$.
  - Compute $\mathbf{X}^T\mathbf{X}$: np^2 flops
  - Compute $\mathbf{X}^T\mathbf{y}$: $2np$ flops
  - Cholesky decomposition of $\mathbf{X}^T\mathbf{X}$: $\frac{1}{3}p^3$ flops
  - Solve normal equation $\mathbf{X}^T\mathbf{X}\beta = \mathbf{X}^T\mathbf{y}$: $2p^2$ flops
  - If need standard errors, another $(4/3)p^3$ flops

Total computational cost is $np^2 + (1/3)p^3$ (without s.e.) or $np^2 + (5/3)p^3$ (with s.e.) flops.

Further reading

- Section 7.7 of Numerical Analysis for Statisticians (http://ucla.worldcat.org/title/numerical-analysis-for-statisticians/oclc/793808354&referer=brief_results) of Kenneth Lange (2010).
- Section II.5.3 of Computational Statistics (http://ucla.worldcat.org/title/computational-statistics/oclc/437345409&referer=brief_results) by James Gentle (2010).
- Section 4.2 of Matrix Computation (http://catalog.library.ucla.edu/vwebv/holdingsInfo?bibId=7122088) by Gene Golub and Charles Van Loan (2013).