$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$PA = LU$$

$$Av = \lambda v, \quad A^{\mathsf{T}}Av = \sigma^2 v$$

Juliaと線形代数の基礎：ソルバー選定とコード性能最適化

王 安清（ワン アンチン）
東北大学　環境科学研究科　福島研
研究分野：プロセスシステム工学
（設計・プロセス最適化）


```
using LinearAlgebra, SparseArrays
x = A \ b
F = cholesky(Symmetric(A))
ldiv!(x, F, b)
```

$$Ax = b$$

$$\begin{bmatrix} B_1 & C_1 & & \\ A_2 & B_2 & C_2 & \\ & \ddots & \ddots & \ddots \\ & & A_n & B_n \end{bmatrix}$$

JuliaLang Japan 2025
2025年12月13日
東京科学大学 大岡山キャンパス

Overview

- Why learn linear solvers?
- How to choose a solver for $\boldsymbol{Ax} = \boldsymbol{b}$
  - $\boldsymbol{A}$ is symmetric/Hermitian
  - $\boldsymbol{A}$ nonsymmetric/non-Hermitian
- Example 1: Thomas algorithm
- Example 2: Block thomas algorithm

Why learn linear solvers?

- Solving linear systems is a fundamental task in engineering: least squares, linear regression, partial differential equation (PDE) models, optimization (KKT systems), nonlinear solvers, etc.
- In practice, we must choose algorithms that exploit the matrix structure and strike a good balance between numerical accuracy and performance.

Why learn linear solvers?

- **Accuracy.**

$$\boldsymbol{A} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}, \mathrm{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

→ Gaussian elimination

$$\boldsymbol{A} = \begin{bmatrix} 1 & 10^{20} \\ 1 & 1 \end{bmatrix}, \mathrm{b} = \begin{bmatrix} 10^{20} \\ 2 \end{bmatrix}$$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 10^{20} \\ 0 & -10^{20} \end{bmatrix}, \mathrm{b} = \begin{bmatrix} 10^{20} \\ -10^{20} \end{bmatrix}$$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 10^{20} \\ 0 & 1 \end{bmatrix}, \mathrm{b} = \begin{bmatrix} 10^{20} \\ 1 \end{bmatrix}$$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathrm{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$\boldsymbol{x} \approx \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ✖ ➡ Gaussian elimination with partial **pivoting** (GEPP)

$$\boldsymbol{x}^* \approx \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Why learn linear solvers?

- **Performance.**

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

```
using LinearAlgebra, BenchmarkTools
n = 10_000
d = fill(2.0, n)
dl = fill(-1.0, n - 1)
du = fill(-1.0, n - 1)
Atri = Tridiagonal(dl, d, du)
b = ones(n)

# Dense LU (blocked GEPP) (O(n³) time, O(n²) memory)
Adense = Matrix(Atri)
@btime $Adense \ $b;
```

969.285 ms (9 allocations: 763.09 MiB)

```
# Tridiagonal solve (O(n) time, O(n) memory)
@btime $Atri \ $b;
```

129.100 μs (20 allocations: 469.31 KiB)

Later, we'll show how to write an algorithm that further reduces the runtime to **44.600 μs** with **zero allocations (0 bytes).**

Choosing a solver for $Ax = b$

- When solving $\boldsymbol{Ax} = \boldsymbol{b}$, the first thing to check is:
  Is $\boldsymbol{A}$ symmetric or Hermitian?

- **Symmetric matrix.**
  A real square matrix that equals its transpose:

$$\boldsymbol{A} = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix}, \quad \boldsymbol{A}^{\mathsf{T}} = \boldsymbol{A}$$

- **Hermitian matrix.**
  A complex square matrix that equals its conjugate transpose:

$$\boldsymbol{B} = \begin{bmatrix} 2 & 1+\mathrm{i} \\ 1-\mathrm{i} & 3 \end{bmatrix}, \quad \boldsymbol{B}^{\mathsf{H}} = \overline{\boldsymbol{B}}^{\mathsf{T}} = \boldsymbol{B}$$

For symmetric/Hermitian matrix A

- Consider whether $\boldsymbol{A}$ is Strictly Positive Definite (SPD):

$$\boldsymbol{A} = \boldsymbol{A}^{\mathsf{T}},\ \boldsymbol{x}^{\mathsf{T}}\boldsymbol{A}\boldsymbol{x} > 0\ \ \forall \boldsymbol{x} \neq \boldsymbol{0}$$

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \qquad \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

- Equivalently:

$$\lambda_i(\boldsymbol{A}) > 0 \ \text{ for all } i;$$

- SPD
  - Dense
    - *Right-looking blocked Cholesky decomposition[1] (BLAS-3)*

```
using LinearAlgebra
x = cholesky(A) \ b
```

  - Sparse
    - Small Scale
      - *Supernodal Cholesky decomposition with AMD reordering[2]*

```
using LinearAlgebra
x = cholesky(A::SparseMatrixCSC) \ b
```

    - Medium–Large Scale
    - ...

[1] Backend: LAPACK-POTRF
[2] Backend: SuiteSparse-CHOLMOD

Large scale sparse SPD

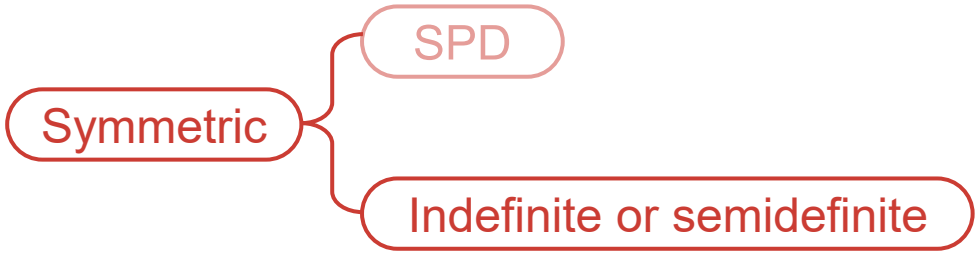
- Conjugate Gradient (CG)

```
using IterativeSolvers
x, [history] = IterativeSolvers.cg(A, b; kwargs...)

using Krylov
x, stats = Krylov.cg(A, b::AbstractVector{FC};)

using LinearSolve # SciML unified ecosystem
sol = solve(LinearProblem(A, b; u0=x0), KrylovJL_CG())
sol = solve(LinearProblem(A, b; u0=x0), IterativeSolversJL_CG())
```

- Optional: build a preconditioner from $\boldsymbol{A}$.
  - Related packages: `Preconditioners.jl`, `IncompleteCholesky.jl`, `LinearAlgebra`
  - Example (Jacobi/diagonal preconditioner)

```
using Krylov, LinearAlgebra
M = Diagonal(diag(A))
x, stats = Krylov.cg(A, b; M)
```

Indefinite or semidefinite



- Small Scale
  - Bunch-Kaufman pivoted $\boldsymbol{LDL}^{\mathsf{T}}$ factorization[1]

```
using LinearAlgebra
x = bunchkaufman(A) \ b
```

- Large Scale
  - MINimum RESidual method (MINRES)

```
using IterativeSolvers
x, [history] = IterativeSolvers.minres(A, b; kwargs...)
using Krylov
x, stats = Krylov.minres(A, b::AbstractVector)
```

[1] Backend: SuiteSparse-CHOLMOD

For nonsymmetric/non-Hermitian matrix A

- Small scale, dense
  - Blocked gaussian elimination with partial pivoting (GEPP) (LU factorization)[1]

```
using LinearAlgebra
x = lu(A::AbstractMatrix) \ b
```

- Small scale, sparse
  - Unsymmetric multifrontal LU factorization with partial pivoting[2]

```
using LinearAlgebra, SparseArrays
x = lu(A::AbstractSparseMatrixCSC) \ b
```

[1] Backend: LAPACK-GETRF
[2] Backend: SuiteSparse-UMFPACK

For nonsymmetric/non-Hermitian matrix A

- Large scale
  - Unsymmetric multifrontal LU factorization with partial pivoting
  - Generalized Minimal Residual (GMRES)

```
using IterativeSolvers
x, [history] = IterativeSolvers.gmres(A, b; kwargs...)

using Krylov
x, stats = Krylov.gmres(A, b)
```

  - Bi-Conjugate Gradient Stabilized (BiCGSTAB)

```
using Krylov
x, stats = Krylov.bicgstab(A, b)
```

For nonsymmetric/non-Hermitian matrix A

- Large scale
  - Quasi-Minimal Residual (QMR) method

```
using IterativeSolvers
x, [history] = IterativeSolvers.qmr(A, b; kwargs...)

using Krylov
x, stats = Krylov.qmr(A, b)
```

  - …
- Preconditioner
  - `IncompleteLU.jl`

Special structures

- SPD banded → Banded Cholesky $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathsf{T}}$

$$\boldsymbol{A} = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}$$

- Symmetric indefinite banded → Banded pivoted $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\mathsf{T}}$

$$\boldsymbol{A} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Nonsymmetric banded → Banded LU with partial pivoting

$$\boldsymbol{A} = \begin{bmatrix} 4 & -1 & 2 & 0 & 0 & 0 \\ -2 & 5 & -1 & 1 & 0 & 0 \\ 0 & -3 & 6 & -1 & 2 & 0 \\ 0 & 0 & -2 & 5 & -1 & 1 \\ 0 & 0 & 0 & -3 & 5 & -1 \\ 0 & 0 & 0 & 0 & -2 & 4 \end{bmatrix}$$

Special structures

- Tridiagonal (general) → Thomas algorithm

$$\boldsymbol{A} = \begin{bmatrix} \beta_1 & \gamma_1 & & \\ \alpha_2 & \beta_2 & \gamma_2 & \\ & \ddots & \ddots & \ddots \\ & & \alpha_n & \beta_n \end{bmatrix}$$

- Block tridiagonal → Block Thomas / block LU

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{C}_1 & & \\ \boldsymbol{A}_2 & \boldsymbol{B}_2 & \boldsymbol{C}_2 & \\ & \ddots & \ddots & \ddots \\ & & \boldsymbol{A}_m & \boldsymbol{B}_m \end{bmatrix}$$

- Cyclic tridiagonal → Sherman–Morrison (SMW) or Cyclic Reduction (CR)

$$\boldsymbol{A} = \begin{bmatrix} \beta_1 & \gamma_1 & 0 & \cdots & \alpha_1 \\ \alpha_2 & \beta_2 & \gamma_2 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ \gamma_n & 0 & \cdots & \alpha_n & \beta_n \end{bmatrix}$$

Example 1: Thomas algorithm

- We solve the tridiagonal system with $a_1 = 0$,$c_n = 0$.

$$\left[\begin{array}{cccccc|c} b_1 & c_1 & 0 & 0 & 0 & 0 & d_1 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & d_2 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & d_3 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & d_4 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & d_5 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & d_6 \end{array}\right]$$

- **Step 1. Forward sweep**
- Eliminate subdiagonal entries.

$$\left[\begin{array}{cccccc|c} 1 & c_1' & 0 & 0 & 0 & 0 & d_1' \\ 0 & \tilde{b}_2 & c_2 & 0 & 0 & 0 & d_2' \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & d_3 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & d_4 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & d_5 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & d_6 \end{array}\right]$$

- **Step 2. Back substitution**
- Now the system is upper triangular in terms of c'_i, d'_i.
- $x_i = d'_i - c'_i x_{i+1}$.

$$\left[\begin{array}{cccccc|c} 1 & c_1' & 0 & 0 & 0 & 0 & d_1' \\ 0 & 1 & c_2' & 0 & 0 & 0 & d_2' \\ 0 & 0 & 1 & c_3' & 0 & 0 & d_3' \\ 0 & 0 & 0 & 1 & c_4' & 0 & d_4' \\ 0 & 0 & 0 & 0 & 1 & c_5' & d_5' \\ 0 & 0 & 0 & 0 & 0 & 1 & d_6' \end{array}\right]$$

Example 1: Thomas algorithm

```
function thomas_tridiagonal(
    L::AbstractVector,  # subdiagonal (a)
    D::AbstractVector,  # diagonal    (b)
    U::AbstractVector,  # superdiag   (c)
    b::AbstractVector   # right-hand side
)
    n    = length(b)
    Upos = Vector{Float64}(undef, n - 1) # modified superdiagonal (c′)
    x    = Vector{Float64}(undef, n)

    # Forward sweep
    Upos[1] = U[1] / D[1]
    x[1]    = b[1] / D[1]
    for i in 2:n-1
        di      = D[i] - L[i] * Upos[i-1]
        Upos[i] = U[i] / di                       # c′_i
        x[i]    = (b[i] - L[i] * x[i-1]) / di     # d′_i
    end
    di   = D[n] - L[n] * Upos[n-1]
    x[n] = (b[n] - L[n] * x[n-1]) / di

    # Back substitution
    for i in n-1:-1:1
        x[i] -= Upos[i] * x[i+1]
    end

    return x
end
```

61.600 μs (6 allocations: 156.39 KiB)

Example 1: Thomas algorithm

```
function thomas_tridiagonal(
    L::AbstractVector,  # subdiagonal (a)
    D::AbstractVector,  # diagonal    (b)
    U::AbstractVector,  # superdiag   (c)
    b::AbstractVector   # right-hand side
)
    n    = length(b)
    Upos = Vector{Float64}(undef, n - 1) # modified superdiagonal (c′)
    x    = Vector{Float64}(undef, n)
    @inbounds begin
        # Forward sweep
        Upos[1] = U[1] / D[1]
        x[1]    = b[1] / D[1]
        for i in 2:n-1
            di      = D[i] - L[i] * Upos[i-1]
            Upos[i] = U[i] / di                      # c′_i
            x[i]    = (b[i] - L[i] * x[i-1]) / di    # d′_i
        end
        di   = D[n] - L[n] * Upos[n-1]
        x[n] = (b[n] - L[n] * x[n-1]) / di

        # Back substitution
        for i in n-1:-1:1
            x[i] -= Upos[i] * x[i+1]
        end
    end
    return x
end
```

61.600 μs (6 allocations: 156.39 KiB)
46.900 μs (5 allocations: 156.36 KiB)

Example 1: Thomas algorithm

```
function thomas_tridiagonal!(x::AbstractVector,
    L::AbstractVector,  # subdiagonal (a)
    D::AbstractVector,  # diagonal    (b)
    U::AbstractVector,  # superdiag   (c)
    b::AbstractVector   # right-hand side
)
    n    = length(b)
    Upos = Vector{Float64}(undef, n - 1) # modified superdiagonal (c')

    @inbounds begin
        # Forward sweep
        Upos[1] = U[1] / D[1]
        x[1]    = b[1] / D[1]
        for i in 2:n-1
            di      = D[i] - L[i] * Upos[i-1]
            Upos[i] = U[i] / di                      # c'_i
            x[i]    = (b[i] - L[i] * x[i-1]) / di    # d'_i
        end
        di   = D[n] - L[n] * Upos[n-1]
        x[n] = (b[n] - L[n] * x[n-1]) / di

        # Back substitution
        for i in n-1:-1:1
            x[i] -= Upos[i] * x[i+1]
        end
    end
    return x
end
```

Need buffer!

Global scratch buffer

```
const Upos = Vector{Float64}(undef, n - 1)
function get_buffers(n::Int)
    length(Upos) != n && resize!(Upos, n)
    return Upos
end
function thomas_tridiagonal!(…)
…
end
```

- Problem: **Not thread safe**
  - Two threads can resize at the same time
  - The buffer can be reallocated while another thread is using it
  - Leads to data races

Task local storage for scratch buffers

```
using TaskLocalValues

const TLS_buf = TaskLocalValue{Vector{Float64}}(() ->
Float64[])

function get_buffers(n::Int)

    buf = TLS_buf[]

    length(buf) != n && resize!(buf, n)

    return buf

end
```

Example 1: Thomas algorithm

```
function thomas_tridiagonal!(x::AbstractVector,
    L::AbstractVector,  # subdiagonal (a)
    D::AbstractVector,  # diagonal    (b)
    U::AbstractVector,  # superdiag   (c)
    b::AbstractVector   # right-hand side
)
    n    = length(b)
    Upos = get_buffers(n - 1) # modified superdiagonal (c')

    @inbounds begin
        # Forward sweep
        Upos[1] = U[1] / D[1]
        x[1]    = b[1] / D[1]
        for i in 2:n-1
            di      = D[i] - L[i] * Upos[i-1]
            Upos[i] = U[i] / di                      # c'_i
            x[i]    = (b[i] - L[i] * x[i-1]) / di    # d'_i
        end
        di   = D[n] - L[n] * Upos[n-1]
        x[n] = (b[n] - L[n] * x[n-1]) / di

        # Back substitution
        for i in n-1:-1:1
            x[i] -= Upos[i] * x[i+1]
        end
    end
    return x
end
```

@simd?

61.600 μs (6 allocations: 156.39 KiB)
46.900 μs (5 allocations: 156.36 KiB)
44.600 μs (0 allocations: 0 bytes)

Example 2: Block thomas algorithm

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{D}_1 & \boldsymbol{U}_1 & & & \\ \boldsymbol{L}_2 & \boldsymbol{D}_2 & \boldsymbol{U}_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \boldsymbol{L}_{n-1} & \boldsymbol{D}_{n-1} & \boldsymbol{U}_{n-1} \\ & & & \boldsymbol{L}_n & \boldsymbol{D}_n \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \vdots \\ \boldsymbol{b}_n \end{bmatrix}$$

$$\tilde{\boldsymbol{U}}_1 = \boldsymbol{D}_1^{-1}\boldsymbol{U}_1,$$
$$\tilde{\boldsymbol{b}}_1 = \boldsymbol{D}_1^{-1}\boldsymbol{b}_1,$$

$$\boldsymbol{S}_i = \boldsymbol{D}_i - \boldsymbol{L}_i\tilde{\boldsymbol{U}}_{i-1},$$
$$\tilde{\boldsymbol{U}}_i = \boldsymbol{S}_i^{-1}\boldsymbol{U}_i,$$
$$\tilde{\boldsymbol{b}}_i = \boldsymbol{S}_i^{-1}\left(\boldsymbol{b}_i - \boldsymbol{L}_i\tilde{\boldsymbol{b}}_{i-1}\right), \qquad i = 2, \ldots, n-1,$$

$$\boldsymbol{S}_n = \boldsymbol{D}_n - \boldsymbol{L}_n\tilde{\boldsymbol{U}}_{n-1},$$
$$\tilde{\boldsymbol{b}}_n = \boldsymbol{S}_n^{-1}\left(\boldsymbol{b}_n - \boldsymbol{L}_n\tilde{\boldsymbol{b}}_{n-1}\right).$$

Example 2: Block thomas algorithm

```
function block_thomas_tridiagonal!(x, L, D, U, b)
    n = size(D[1], 1)
    m = length(D)
    lu_buf = Vector{LU{Float64,Matrix{Float64},Vector{Int64}}}(undef, m - 1)

    # Forward elimination
    @inbounds for i = 1:m-1
        F = lu_buf[i] = lu!(D[i])
        rdiv!(L[i], F)
        # D[i+1] ← -1*L[i]*U[i] + 1*D[i+1]
        # BLAS GEMM: C = α A B + β C
        mul!(D[i+1], L[i], U[i], -1, 1)
        bᵢ   = @view b[(i-1)*n+1:i*n]
        bᵢ₊₁ = @view b[i*n+1:(i+1)*n]
        # b_{i+1} ← b_{i+1} - L[i] * b_i
        # BLAS GEMV: y = α A x + β y
        mul!(bᵢ₊₁, L[i], bᵢ, -1, 1)
    end
```

$$\tilde{\boldsymbol{U}}_1 \leftarrow \boldsymbol{D}_1^{-1}\boldsymbol{U}_1,$$
$$\tilde{\boldsymbol{b}}_1 \leftarrow \boldsymbol{D}_1^{-1}\boldsymbol{b}_1,$$
$$\boldsymbol{D}_{i+1} \leftarrow \boldsymbol{D}_{i+1} - \boldsymbol{L}_i\tilde{\boldsymbol{U}}_i,$$
$$\tilde{\boldsymbol{U}}_{i+1} \leftarrow \boldsymbol{D}_{i+1}^{-1}\boldsymbol{U}_{i+1},$$
$$\tilde{\boldsymbol{b}}_{i+1} \leftarrow \boldsymbol{D}_{i+1}^{-1}\left(\boldsymbol{b}_{i+1} - \boldsymbol{L}_i\tilde{\boldsymbol{b}}_i\right),$$
$$\text{for } i = 1, \ldots, n-1.$$

Example 2: Block thomas algorithm

```
    # Backward substitution (D[i]*x_i = b_i - U[i]*x_{i+1})
    Fm = lu!(D[m])
    xm = @view x[(m-1)*n+1:m*n]
    bm = @view b[(m-1)*n+1:m*n]
    ldiv!(xm, Fm, bm)
    @inbounds for i = m-1:-1:1
        x_i   = @view x[(i-1)*n+1:i*n]
        rhs   = @view b[(i-1)*n+1:i*n]
        x_{i+1} = @view x[i*n+1:(i+1)*n]
        ldiv!(x_i, lu_buf[i], mul!(rhs, U[i], x_{i+1}, -1, 1))
    end

    return x
end
```

Conclusions

- **Solving linear systems**
  - Check matrix structure first (tridiagonal, block, banded, sparse, …)
  - Choose an algorithm that matches the structure
  - Then pick the Julia library / implementation
  - Add a preconditioner when necessary
- **Writing your own solver**
  - A custom implementation can sometimes outperform general-purpose libraries.
  - Prefer **in-place, mutating operations** in hot loops.
  - Use **task-local scratch buffers** to eliminate allocations while remaining thread-safe.
  - Whenever possible, **rewrite the mathematics into BLAS-friendly kernels**, rather than copying formulas line by line into code.

ご清聴ありがとうございました
Thank you for your attention

王 安清
GitHub: @abcdvvvv

Scan to download the slides (PDF) ⟶