Linear algebra

Solves, factorisations, wrapper types

Half the reason the language exists. `LinearAlgebra` is a stdlib, so `using LinearAlgebra` needs no installation.

Basics

```
using LinearAlgebra

A = rand(3, 3); B = rand(3, 3); v = rand(3)

A * B              # matrix multiply
A .* B             # elementwise
A * v              # matrix-vector
v' * v             # inner product (scalar); v' is a lazy adjoint
v * v'             # outer product (3x3)
A'                 # adjoint (conjugate transpose), lazy
transpose(A)       # transpose without conjugation, lazy
A^2                # matrix power, not elementwise
A.^2               # elementwise
```

Solving

```
x = A \ b          # solve A x = b
x = A' \ b
X = A \ B          # multiple right-hand sides
```

`\` inspects the matrix, picks an appropriate factorisation (LU, Cholesky, QR for over-determined, least-squares for non-square), and dispatches to LAPACK. **Never write `inv(A) * b`** — slower, less accurate, and it computes something you didn't need.

For an over-determined system, `A \ b` gives the least-squares solution directly.

Reuse a factorisation across multiple solves:

```
F = lu(A)
x1 = F \ b1
x2 = F \ b2

F = cholesky(Symmetric(A))     # if A is symmetric positive definite
F = qr(A)
```

Factorisations

```
lu(A); qr(A); cholesky(A); ldlt(A)
eigen(A)                       # returns an Eigen; F.values, F.vectors
eigvals(A); eigvecs(A)
svd(A)                         # F.U, F.S, F.V, F.Vt
schur(A); hessenberg(A)
bunchkaufman(A)
pinv(A)                        # Moore-Penrose pseudoinverse
```

Factorisation objects are lazy wrappers: `F \ b` uses the factors directly, `Matrix(F)` materialises.

Scalars from matrices

```
tr(A)              # trace
det(A); logdet(A); logabsdet(A)     # use logdet for likelihoods; det underflows
rank(A)
cond(A)            # condition number
norm(v); norm(v, 1); norm(v, Inf); opnorm(A, 2)
dot(u, v); u ⋅ v   # \cdot<tab>
cross(u, v); u × v # \times<tab>
```

Constructors and wrappers

```
I                          # UniformScaling: A + 2I works at any size, allocates nothing
Matrix(I, 3, 3)            # a dense identity when you actually need one
Diagonal([1,2,3])
Bidiagonal, Tridiagonal, SymTridiagonal
Symmetric(A), Hermitian(A)
UpperTriangular(A), LowerTriangular(A), UnitUpperTriangular(A)
diagm(0 => [1,2,3], 1 => [4,5])
diag(A); diag(A, 1)
```

These wrappers are the single highest-leverage thing in the package. Wrapping is not documentation — it **dispatches to a different algorithm**. `Symmetric(A) \ b` uses Bunch-Kaufman or Cholesky rather than general LU; `Diagonal(d) * A` is $O(n^2)$ instead of $O(n^3)$; `UpperTriangular(A) \ b` is a back-substitution. Wrapping costs nothing and can change the complexity class.

In-place operations

```
mul!(C, A, B)              # C = A*B, no allocation
mul!(C, A, B, α, β)        # C = α*A*B + β*C  (the full BLAS gemm)
ldiv!(F, b)                # b = F \ b
lmul!(2, v); rmul!(v, 2)
axpy!(α, x, y)             # y = α*x + y
copyto!(dest, src)
lu!(A); cholesky!(A)       # factorise in place, destroying A
```

In an iterative algorithm, preallocating and using `mul!` is usually the difference between allocating gigabytes and allocating nothing.

Sparse

```
using SparseArrays
S = sparse(I, J, V, m, n)          # from COO triplets
S = spzeros(1000, 1000)
S = sprand(1000, 1000, 0.01)
sparse(A); Matrix(S)               # convert both ways
nnz(S); nonzeros(S); rowvals(S); findnz(S)
droptol!(S, 1e-12); dropzeros!(S)

S \ b                              # sparse LU (UMFPACK) automatically
cholesky(S)                        # CHOLMOD for SPD sparse
```

Storage is CSC (compressed sparse *column*), consistent with the column-major convention. Building a sparse matrix by assigning into `spzeros` in a loop is pathologically slow — accumulate `I`, `J`, `V` vectors and call `sparse(I, J, V)` once, which also sums duplicate entries for you (exactly what finite-element assembly wants).

Iterative solvers and beyond

For large or matrix-free problems:

- **`IterativeSolvers.jl`** / **`Krylov.jl`** — CG, GMRES, MINRES, LSQR; take any object supporting `mul!`.
- **`LinearSolve.jl`** — a uniform interface over direct and iterative solvers, with algorithm selection and caching. The modern default for library code.
- **`LinearMaps.jl`** — represent a linear operator by its action, no matrix stored.
- **`Preconditioners.jl`**, **`IncompleteLU.jl`** — preconditioning.
- **`ArnoldiMethod.jl`** / **`KrylovKit.jl`** — a few eigenvalues of a large sparse matrix.
- **`StaticArrays.jl`** — `SMatrix{3,3}`, stack-allocated, with hand-unrolled small-matrix algebra. For 3-D geometry in a hot loop this is 10x.
- **`CUDA.jl`** / **`AMDGPU.jl`** — `CuArray` implements the same interfaces, so `A \ b`, `mul!`, broadcasting and most generic code work unchanged on device arrays.

BLAS control

```
BLAS.get_num_threads(); BLAS.set_num_threads(4)
BLAS.vendor()
```

trap: BLAS runs its own thread pool, independent of `JULIA_NUM_THREADS`. Running `Threads.@threads` over matrix operations oversubscribes the CPU badly — each Julia thread launches a multithreaded BLAS call. Set `BLAS.set_num_threads(1)` when you're parallelising at the Julia level.

Numerical notes

```julia
logdet(A)                    # not log(det(A))
A \ b                        # not inv(A) * b
issymmetric(A), isposdef(A), ishermitian(A)
normalize(v); normalize!(v)
nullspace(A); rank(A; rtol = ...)
kron(A, B)                   # Kronecker product
```

Floating-point comparisons of matrices: `A ≈ B` with `isapprox(A, B; rtol = ...)`, never `==`.