Linear Algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with `using LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1

julia> tr(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Array{Float64,2}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

As well as other useful operations, such as finding eigenvalues or eigenvectors:

```
julia> A = [-4. -17.; 2. 2.]
2×2 Array{Float64,2}:
 -4.0  -17.0
  2.0    2.0

julia> eigvals(A)
2-element Array{Complex{Float64},1}:
 -1.0 - 5.0im
 -1.0 + 5.0im

julia> eigvecs(A)
2×2 Array{Complex{Float64},2}:
  0.945905-0.0im        0.945905+0.0im
```

```
 -0.166924+0.278207im  -0.166924-0.278207im
```

In addition, Julia provides many factorizations which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. See the documentation on `factorize` for more information. As an example:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
  1.0    0.0       0.0
 -0.15   1.0       0.0
 -0.3   -0.132196  1.0
U factor:
3×3 Array{Float64,2}:
 -10.0  2.3     4.0
   0.0  2.345  -3.4
   0.0  0.0    -5.24947
```

Since `A` is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
3×3 Tridiagonal{Float64,Array{Float64,1}}:
 -1.64286   0.0   ⋅
  0.0      -2.8  0.0
   ⋅        0.0  5.0
U factor:
```

```
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.142857  -0.8
  ⋅   1.0       -0.6
  ⋅    ⋅         1.0
permutation:
3-element Array{Int64,1}:
 1
 2
 3
```

Here, Julia was able to detect that `B` is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for a matrix that is known to have certain properties e.g. it is symmetric, or tridiagonal. Julia provides some special types so that you can "tag" matrices as having these properties. For instance:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

`sB` has been tagged as a matrix that's (real) symmetric, so for later operations we might perform on it, such as eigenfactorization or computing matrix-vector products, efficiencies can be found by only referencing half of it. For example:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

```
julia> x = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> sB\x
3-element Array{Float64,1}:
 -1.7391304347826084
 -1.1086956521739126
 -1.4565217391304346
```

The `\` operation here performs the linear solution. The left-division operator is pretty powerful and it's easy to write compact, readable code that is flexible enough to solve all sorts of systems of linear equations.

Special matrices

Matrices with special symmetries and structures arise often in linear algebra and are frequently associated with various matrix factorizations. Julia features a rich collection of special matrix types, which allow for fast computation with specialized routines that are specially developed for particular matrix types.

The following tables summarize the types of special matrices that have been implemented in Julia, as well as whether hooks to various optimized methods for them in LAPACK are available.

Type	Description
`Symmetric`	Symmetric matrix
`Hermitian`	Hermitian matrix
`UpperTriangular`	Upper triangular matrix
`UnitUpperTriangular`	Upper triangular matrix with unit diagonal
`LowerTriangular`	Lower triangular matrix
`UnitLowerTriangular`	Lower triangular matrix with unit diagonal
`UpperHessenberg`	Upper Hessenberg matrix
`Tridiagonal`	Tridiagonal matrix

`SymTridiagonal`	Symmetric tridiagonal matrix
`Bidiagonal`	Upper/lower bidiagonal matrix
`Diagonal`	Diagonal matrix
`UniformScaling`	Uniform scaling operator

Elementary operations

Matrix type	`+`	`-`	`*`	`\`	Other functions with optimized methods
`Symmetric`				MV	`inv`, `sqrt`, `exp`
`Hermitian`				MV	`inv`, `sqrt`, `exp`
`UpperTriangular`			MV	MV	`inv`, `det`
`UnitUpperTriangular`			MV	MV	`inv`, `det`
`LowerTriangular`			MV	MV	`inv`, `det`
`UnitLowerTriangular`			MV	MV	`inv`, `det`
`UpperHessenberg`				MM	`inv`, `det`
`SymTridiagonal`	M	M	MS	MV	`eigmax`, `eigmin`
`Tridiagonal`	M	M	MS	MV	
`Bidiagonal`	M	M	MS	MV	
`Diagonal`	M	M	MV	MV	`inv`, `det`, `logdet`, `/`
`UniformScaling`	M	M	MVS	MVS	`/`

Legend:

Key	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available

S (scalar)	An optimized method for matrix-scalar operations is available

Matrix factorizations

Matrix type	LAPACK	`eigen`	`eigvals`	`eigvecs`	`svd`	`svdvals`
`Symmetric`	SY		ARI			
`Hermitian`	HE		ARI			
`UpperTriangular`	TR	A	A	A		
`UnitUpperTriangular`	TR	A	A	A		
`LowerTriangular`	TR	A	A	A		
`UnitLowerTriangular`	TR	A	A	A		
`SymTridiagonal`	ST	A	ARI	AV		
`Tridiagonal`	GT					
`Bidiagonal`	BD				A	A
`Diagonal`	DI		A			

Legend:

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. `eigvals(M)`
R (range)	An optimized method to find the `il`th through the `ih`th characteristic values are available	`eigvals(M, il, ih)`
I (interval)	An optimized method to find the characteristic values in the interval [`vl`, `vh`] is available	`eigvals(M, vl, vh)`
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values `x=[x1, x2,...]` is available	`eigvecs(M, x)`

The uniform scaling operator

A `UniformScaling` operator represents a scalar times the identity operator, `λ*I`. The identity operator `I` is defined as a constant and is an instance of `UniformScaling`. The size of these operators are generic and match the other matrix in the binary operations `+`, `-`, `*` and `\`. For `A+I` and `A-I` this means that `A` must be square. Multiplication with the identity operator `I` is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

To see the `UniformScaling` operator in action:

```
julia> U = UniformScaling(2);

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> a + U
2×2 Array{Int64,2}:
 3  2
 3  6

julia> a * U
2×2 Array{Int64,2}:
 2  4
 6  8

julia> [a U]
2×4 Array{Int64,2}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> b - U
ERROR: DimensionMismatch("matrix is not square: dimensions are (2, 3)")
Stacktrace:
[...]
```

If you need to solve many systems of the form `(A+μI)x = b` for the same `A` and different `μ`, it might be beneficial to first compute the Hessenberg factorization `F` of `A` via the `hessenberg` function. Given `F`, Julia employs an efficient algorithm for `(F+μ*I) \ b` (equivalent to `(A+μ*I)x \ b`) and related operations like determinants.

Matrix factorizations

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the Standard functions section of the Linear Algebra documentation.

Type	Description
`BunchKaufman`	Bunch-Kaufman factorization
`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LDLt`	LDL(T) factorization
`LU`	LU factorization
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`LQ`	QR factorization of `transpose(A)`
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`GeneralizedEigen`	Generalized spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD
`Schur`	Schur decomposition

`GeneralizedSchur`	Generalized Schur decomposition

Standard functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

`Base.:*` — Method

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Array{Int64,2}:
 2  1
 1  1
```

`Base.:\` — Method

```
\(A, B)
```

Matrix division using a polyalgorithm. For input matrices `A` and `B`, the result `X` is such that `A*X == B` when `A` is square. The solver that is used depends upon the structure of `A`. If `A` is upper or lower triangular (or diagonal), no factorization of `A` is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.

For rectangular `A` the result is the minimum-norm least squares solution computed by a pivoted QR factorization of `A` and a rank estimate of `A` based on the R factor.

When `A` is sparse, a similar polyalgorithm is used. For indefinite matrices, the `LDLt` factorization does not use pivoting during the numerical factorization and therefore the procedure can fail even for invertible matrices.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];

julia> X = A \ B
2-element Array{Float64,1}:
 32.0
 18.0

julia> A * X == B
true
```

`LinearAlgebra.SingularException` — Type

```
SingularException
```

Exception thrown when the input matrix has one or more zero-valued eigenvalues, and is not invertible. A linear solve involving such a matrix cannot be computed. The `info` field indicates the location of (one of) the singular value(s).

`LinearAlgebra.PosDefException` — Type

```
PosDefException
```

Exception thrown when the input matrix was not positive definite. Some linear algebra functions and factorizations are only applicable to positive definite matrices. The `info` field indicates the location of (one of) the eigenvalue(s) which is (are) less than/equal to 0.

`LinearAlgebra.ZeroPivotException` — Type

```
ZeroPivotException <: Exception
```

Exception thrown when a matrix factorization/solve encounters a zero in a pivot (diagonal) position and cannot proceed. This may *not* mean that the matrix is singular: it may be fruitful to switch to a diffent factorization such as pivoted LU that can re-order variables to eliminate spurious zero pivots. The `info` field indicates the location of (one of) the zero pivot(s).

`LinearAlgebra.dot` — Function

```
dot(x, y)
x ⋅ y
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated.

`dot` also works on arbitrary iterable objects, including arrays of any dimension, as long as `dot` is defined on the elements.

`dot` is semantically equivalent to `sum(dot(vx,vy) for (vx,vy) in zip(x, y))`, with the added restriction that the arguments must have equal lengths.

`x ⋅ y` (where `⋅` can be typed by tab-completing `\cdot` in the REPL) is a synonym for `dot(x, y)`.

Examples

```
julia> dot([1; 1], [2; 3])
5

julia> dot([im; im], [1; 1])
0 - 2im

julia> dot(1:5, 2:6)
70

julia> x = fill(2., (5,5));

julia> y = fill(3., (5,5));

julia> dot(x, y)
150.0
```

`LinearAlgebra.cross` — Function

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]
3-element Array{Int64,1}:
 0
 1
 0

julia> b = [0;0;1]
3-element Array{Int64,1}:
 0
 0
 1

julia> cross(a,b)
3-element Array{Int64,1}:
 1
 0
 0
```

`LinearAlgebra.factorize` — Function

```
factorize(A)
```

Compute a convenient factorization of `A`, based upon the type of the input matrix. `factorize` checks `A` to see if it is symmetric/triangular/etc. if `A` is passed as a generic matrix. `factorize` checks every element of `A` to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C`.

Properties of `A`	type of factorization
Positive-definite	Cholesky (see `cholesky`)
Dense Symmetric/Hermitian	Bunch-Kaufman (see `bunchkaufman`)
Sparse Symmetric/Hermitian	LDLt (see `ldlt`)
Triangular	Triangular
Diagonal	Diagonal

Bidiagonal	Bidiagonal
Tridiagonal	LU (see `lu`)
Symmetric real tridiagonal	LDLt (see `ldlt`)
General square	LU (see `lu`)
General non-square	QR (see `qr`)

If `factorize` is called on a Hermitian positive-definite matrix, for instance, then `factorize` will return a Cholesky factorization.

Examples

```
julia> A = Array(Bidiagonal(fill(1.0, (5, 5)), :U))
5×5 Array{Float64,2}:
 1.0  1.0  0.0  0.0  0.0
 0.0  1.0  1.0  0.0  0.0
 0.0  0.0  1.0  1.0  0.0
 0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0

julia> factorize(A) # factorize will check to see that A is already factorized
5×5 Bidiagonal{Float64,Array{Float64,1}}:
 1.0  1.0   ⋅    ⋅    ⋅
  ⋅   1.0  1.0   ⋅    ⋅
  ⋅    ⋅   1.0  1.0   ⋅
  ⋅    ⋅    ⋅   1.0  1.0
  ⋅    ⋅    ⋅    ⋅   1.0
```

This returns a `5×5 Bidiagonal{Float64}`, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for `Bidiagonal` types.

`LinearAlgebra.Diagonal` — Type

```
Diagonal(A::AbstractMatrix)
```

Construct a matrix from the diagonal of `A`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> Diagonal(A)
3×3 Diagonal{Int64,Array{Int64,1}}:
 1  ⋅  ⋅
 ⋅  5  ⋅
 ⋅  ⋅  9
```

```
Diagonal(V::AbstractVector)
```

Construct a matrix with `V` as its diagonal.

Examples

```
julia> V = [1, 2]
2-element Array{Int64,1}:
 1
 2

julia> Diagonal(V)
2×2 Diagonal{Int64,Array{Int64,1}}:
 1  ⋅
 ⋅  2
```

`LinearAlgebra.Bidiagonal` — Type

```
Bidiagonal(dv::V, ev::V, uplo::Symbol) where V <: AbstractVector
```

Constructs an upper (`uplo=:U`) or lower (`uplo=:L`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The length of `ev` must be one less than the length of `dv`.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first superdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  7  ⋅  ⋅
 ⋅  2  8  ⋅
 ⋅  ⋅  3  9
 ⋅  ⋅  ⋅  4

julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  ⋅  ⋅  ⋅
 7  2  ⋅  ⋅
 ⋅  8  3  ⋅
 ⋅  ⋅  9  4
```

```
Bidiagonal(A, uplo::Symbol)
```

Construct a `Bidiagonal` matrix from the main diagonal of `A` and its first super- (if `uplo=:U`) or sub-diagonal (if `uplo=:L`).

Examples

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Array{Int64,2}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, :U) # contains the main diagonal and first superdiagonal c
```

```
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  1  ⋅  ⋅
 ⋅  2  2  ⋅
 ⋅  ⋅  3  3
 ⋅  ⋅  ⋅  4

julia> Bidiagonal(A, :L) # contains the main diagonal and first subdiagonal of
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  ⋅  ⋅  ⋅
 2  2  ⋅  ⋅
 ⋅  3  3  ⋅
 ⋅  ⋅  4  4
```

`LinearAlgebra.SymTridiagonal` — Type

```
SymTridiagonal(dv::V, ev::V) where V <: AbstractVector
```

Construct a symmetric tridiagonal matrix from the diagonal (`dv`) and first sub/super-diagonal (`ev`), respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

For `SymTridiagonal` block matrices, the elements of `dv` are symmetrized. The argument `ev` is interpreted as the superdiagonal. Blocks from the subdiagonal are (materialized) transpose of the corresponding superdiagonal blocks.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
```

```
4×4 SymTridiagonal{Int64,Array{Int64,1}}:
 1  7  ⋅  ⋅
 7  2  8  ⋅
 ⋅  8  3  9
 ⋅  ⋅  9  4

julia> A = SymTridiagonal(fill([1 2; 3 4], 3), fill([1 2; 3 4], 2));

julia> A[1,1]
2×2 Symmetric{Int64,Array{Int64,2}}:
 1  2
 2  4

julia> A[1,2]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A[2,1]
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first superdiagonal of the symmetric matrix `A`.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Array{Int64,2}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64,Array{Int64,1}}:
 1  2  ⋅
 2  4  5
 ⋅  5  6

julia> B = reshape([[1 2; 2 3], [1 2; 3 4], [1 3; 2 4], [1 2; 2 3]], 2, 2);
```

```
julia> SymTridiagonal(B)
2×2 SymTridiagonal{Array{Int64,2},Array{Array{Int64,2},1}}:
 [1 2; 2 3]  [1 3; 2 4]
 [1 2; 3 4]  [1 2; 2 3]
```

`LinearAlgebra.Tridiagonal` — Type

```
Tridiagonal(dl::V, d::V, du::V) where V <: AbstractVector
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Examples

```
julia> dl = [1, 2, 3];

julia> du = [4, 5, 6];

julia> d = [7, 8, 9, 0];

julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 7  4  ⋅  ⋅
 1  8  5  ⋅
 ⋅  2  9  6
 ⋅  ⋅  3  0
```

```
Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix `A`.

Examples

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Array{Int64,2}:
```

```
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4

julia> Tridiagonal(A)
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 1  2  ⋅  ⋅
 1  2  3  ⋅
 ⋅  2  3  4
 ⋅  ⋅  3  4
```

`LinearAlgebra.Symmetric` — Type

```
Symmetric(A, uplo=:U)
```

Construct a `Symmetric` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Examples

```
julia> A = [1 0 2 0 3; 0 4 0 5 0; 6 0 7 0 8; 0 9 0 1 0; 2 0 3 0 4]
5×5 Array{Int64,2}:
 1  0  2  0  3
 0  4  0  5  0
 6  0  7  0  8
 0  9  0  1  0
 2  0  3  0  4

julia> Supper = Symmetric(A)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  2  0  3
 0  4  0  5  0
 2  0  7  0  8
 0  5  0  1  0
 3  0  8  0  4

julia> Slower = Symmetric(A, :L)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  6  0  2
 0  4  0  9  0
 6  0  7  0  3
```

```
 0  9  0  1  0
 2  0  3  0  4
```

Note that `Supper` will not be equal to `Slower` unless `A` is itself symmetric (e.g. if `A == transpose(A)`).

`LinearAlgebra.Hermitian` — Type

```
Hermitian(A, uplo=:U)
```

Construct a `Hermitian` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Examples

```
julia> A = [1 0 2+2im 0 3-3im; 0 4 0 5 0; 6-6im 0 7 0 8+8im; 0 9 0 1 0; 2+2im 0

julia> Hupper = Hermitian(A)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  2+2im  0+0im  3-3im
 0+0im  4+0im  0+0im  5+0im  0+0im
 2-2im  0+0im  7+0im  0+0im  8+8im
 0+0im  5+0im  0+0im  1+0im  0+0im
 3+3im  0+0im  8-8im  0+0im  4+0im

julia> Hlower = Hermitian(A, :L)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  6+6im  0+0im  2-2im
 0+0im  4+0im  0+0im  9+0im  0+0im
 6-6im  0+0im  7+0im  0+0im  3+3im
 0+0im  9+0im  0+0im  1+0im  0+0im
 2+2im  0+0im  3-3im  0+0im  4+0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if `A == adjoint(A)`).

All non-real parts of the diagonal will be ignored.

```
Hermitian(fill(complex(1,1), 1, 1)) == fill(1, 1, 1)
```

`LinearAlgebra.LowerTriangular` — Type

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> LowerTriangular(A)
3×3 LowerTriangular{Float64,Array{Float64,2}}:
 1.0   ⋅    ⋅
 4.0  5.0   ⋅
 7.0  8.0  9.0
```

`LinearAlgebra.UpperTriangular` — Type

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UpperTriangular(A)
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0  3.0
  ⋅   5.0  6.0
  ⋅    ⋅   9.0
```

`LinearAlgebra.UnitLowerTriangular` — Type

```
UnitLowerTriangular(A::AbstractMatrix)
```

Construct a `UnitLowerTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitLowerTriangular(A)
3×3 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0   ⋅    ⋅
 4.0  1.0   ⋅
 7.0  8.0  1.0
```

`LinearAlgebra.UnitUpperTriangular` — Type

```
UnitUpperTriangular(A::AbstractMatrix)
```

Construct an `UnitUpperTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitUpperTriangular(A)
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0  3.0
  ⋅   1.0  6.0
  ⋅    ⋅   1.0
```

`LinearAlgebra.UpperHessenberg` — Type

```
UpperHessenberg(A::AbstractMatrix)
```

Construct an `UpperHessenberg` view of the matrix `A`. Entries of `A` below the first subdiagonal are ignored.

Efficient algorithms are implemented for `H \ b`, `det(H)`, and similar.

See also the `hessenberg` function to factor any matrix into a similar upper-Hessenberg matrix.

If `F::Hessenberg` is the factorization object, the unitary matrix can be accessed with `F.Q` and the Hessenberg matrix with `F.H`. When `Q` is extracted, the resulting type is the `HessenbergQ` object, and may be converted to a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Iterating the decomposition produces the factors `F.Q` and `F.H`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
4×4 Array{Int64,2}:
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16

julia> UpperHessenberg(A)
4×4 UpperHessenberg{Int64,Array{Int64,2}}:
 1   2   3   4
 5   6   7   8
 ⋅  10  11  12
 ⋅   ⋅  15  16
```

`LinearAlgebra.UniformScaling` — Type

```
UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator, `λ*I`. See also `I`.

Examples

```
julia> J = UniformScaling(2.)
UniformScaling{Float64}
2.0*I

julia> A = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> J*A
2×2 Array{Float64,2}:
 2.0  4.0
 6.0  8.0
```

`LinearAlgebra.I` — Constant

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Examples

```
julia> fill(1, (5,6)) * I == fill(1, (5,6))
true

julia> [1 2im 3; 1im 2 3] * I
2×3 Array{Complex{Int64},2}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

`LinearAlgebra.Factorization` — Type

```
LinearAlgebra.Factorization
```

Abstract type for matrix factorizations a.k.a. matrix decompositions. See online documentation for a list of available matrix factorizations.

`LinearAlgebra.LU` — Type

```
LU <: Factorization
```

Matrix factorization type of the `LU` factorization of a square matrix `A`. This is the return type of `lu`, the corresponding matrix factorization function.

The individual components of the factorization `F::LU` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (unit lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`

`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

Examples

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu` — Function

```
lu(A, pivot=Val(true); check = true) -> F::LU
```

Compute the LU factorization of `A`.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

In most cases, if `A` is a subtype `S` of `AbstractMatrix{T}` with an element type `T` supporting `+`, `-`, `*` and `/`, the return type is `LU{T,S{T}}`. If pivoting is chosen (default) the element type should also support `abs` and `<`.

The individual components of the factorization `F` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

The relationship between `F` and `A` is

`F.L*F.U == A[F.p, :]`

`F` further supports the following functions:

Supported function	`LU`	`LU{T,Tridiagonal{T}}`
`/`	✓	
`\`	✓	✓
`inv`	✓	✓
`det`	✓	✓
`logdet`	✓	✓
`logabsdet`	✓	✓
`size`	✓	✓

Examples

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3
```

```
julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu!` — Function

```
lu!(A, pivot=Val(true); check = true) -> LU
```

`lu!` is the same as `lu`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Array{Float64,2}:
 4.0  3.0
 6.0  3.0

julia> F = lu!(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
```

```
 6.0  3.0
 0.0  1.0

julia> iA = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> lu!(iA)
ERROR: InexactError: Int64(0.6666666666666666)
Stacktrace:
[...]
```

`LinearAlgebra.Cholesky` — Type

```
Cholesky <: Factorization
```

Matrix factorization type of the Cholesky factorization of a dense symmetric/Hermitian positive definite matrix `A`. This is the return type of `cholesky`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::Cholesky` via `F.L` and `F.U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
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

julia> C.U
```

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

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ⋅    ⋅
  6.0  1.0   ⋅
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

`LinearAlgebra.CholeskyPivoted` — Type

```
CholeskyPivoted
```

Matrix factorization type of the pivoted Cholesky factorization of a dense symmetric/Hermitian positive semi-definite matrix `A`. This is the return type of `cholesky(_, Val(true))`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::CholeskyPivoted` via `F.L` and `F.U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A, Val(true))
CholeskyPivoted{Float64,Array{Float64,2}}
U factor with rank 3:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 9.89949  -4.34366  -1.61624
  ⋅        4.25825   1.1694
  ⋅         ⋅        0.142334
permutation:
```

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

`LinearAlgebra.cholesky` — Function

```
cholesky(A, Val(false); check = true) -> Cholesky
```

Compute the Cholesky factorization of a dense symmetric positive definite matrix `A` and return a `Cholesky` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian` `StridedMatrix` or a *perfectly* symmetric or Hermitian `StridedMatrix`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F.L` and `F.U`. The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
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

julia> C.U
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
```

```
  ⋅    ⋅    3.0

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ⋅    ⋅
  6.0  1.0   ⋅
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

```
cholesky(A, Val(true); tol = 0.0, check = true) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix `A` and return a `CholeskyPivoted` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian` `StridedMatrix` or a *perfectly* symmetric or Hermitian `StridedMatrix`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F.L` and `F.U`. The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`. The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

`LinearAlgebra.cholesky!` — Function

```
cholesky!(A, Val(false); check = true) -> Cholesky
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
```

```
 1   2
 2  50

julia> cholesky!(A)
ERROR: InexactError: Int64(6.782329983125268)
Stacktrace:
[...]
```

```
cholesky!(A, Val(true); tol = 0.0, check = true) -> CholeskyPivoted
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

`LinearAlgebra.lowrankupdate` — Function

```
lowrankupdate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U + v*v')` but the computation of `CC` only uses `O(n^2)` operations.

`LinearAlgebra.lowrankdowndate` — Function

```
lowrankdowndate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U - v*v')` but the computation of `CC` only uses `O(n^2)` operations.

`LinearAlgebra.lowrankupdate!` — Function

```
lowrankupdate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U + v*v')` but the computation of `CC` only uses `O(n^2)` operations. The input factorization `C` is updated in place such that on exit `C == CC`. The vector `v` is destroyed during the computation.

`LinearAlgebra.lowrankdowndate!` — Function

```
lowrankdowndate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U - v*v')` but the computation of `CC` only uses `O(n^2)` operations. The input factorization `C` is updated in place such that on exit `C == CC`. The vector `v` is destroyed during the computation.

`LinearAlgebra.LDLt` — Type

```
LDLt <: Factorization
```

Matrix factorization type of the `LDLt` factorization of a real `SymTridiagonal` matrix `S` such that `S = L*Diagonal(d)*L'`, where `L` is a `UnitLowerTriangular` matrix and `d` is a vector. The main use of an `LDLt` factorization `F = ldlt(S)` is to solve the linear system of equations `Sx = b` with `F\b`. This is the return type of `ldlt`, the corresponding matrix factorization function.

The individual components of the factorization `F::LDLt` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (unit lower triangular) part of `LDLt`
`F.D`	`D` (diagonal) part of `LDLt`
`F.Lt`	`Lt` (unit upper triangular) part of `LDLt`
`F.d`	diagonal values of `D` as a `Vector`

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ⋅
```

```
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> F = ldlt(S)
LDLt{Float64,SymTridiagonal{Float64,Array{Float64,1}}}
L factor:
3×3 UnitLowerTriangular{Float64,SymTridiagonal{Float64,Array{Float64,1}}}:
 1.0        ⋅         ⋅
 0.333333  1.0        ⋅
 0.0       0.545455  1.0
D factor:
3×3 Diagonal{Float64,Array{Float64,1}}:
 3.0   ⋅        ⋅
  ⋅   3.66667   ⋅
  ⋅    ⋅       3.90909
```

`LinearAlgebra.ldlt` — Function

```
ldlt(S::SymTridiagonal) -> LDLt
```

Compute an `LDLt` factorization of the real symmetric tridiagonal matrix `S` such that `S = L*Diagonal(d)*L'` where `L` is a unit lower triangular matrix and `d` is a vector. The main use of an `LDLt` factorization `F = ldlt(S)` is to solve the linear system of equations `Sx = b` with `F\b`.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> ldltS = ldlt(S);

julia> b = [6., 7., 8.];

julia> ldltS \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

```
julia> S \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

`LinearAlgebra.ldlt!` — Function

```
ldlt!(S::SymTridiagonal) -> LDLt
```

Same as `ldlt`, but saves space by overwriting the input `S`, instead of creating a copy.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> ldltS = ldlt!(S);

julia> ldltS === S
false

julia> S
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0       0.333333   ⋅
 0.333333  3.66667   0.545455
  ⋅        0.545455  3.90909
```

`LinearAlgebra.QR` — Type

```
QR <: Factorization
```

A QR matrix factorization stored in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.

`LinearAlgebra.QRCompactWY` — Type

```
QRCompactWY <: Factorization
```

A QR matrix factorization stored in a compact blocked format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the `QR` format except that the orthogonal/unitary matrix Q is stored in *Compact WY* format [Schreiber1989]. For the block size n_b, it is stored as a `m×n` lower trapezoidal matrix V and a matrix $T = (T_1 \; T_2 \; ... \; T_{b-1} \; T_b')$ composed of $b = \lceil \min(m,n)/n_b \rceil$ upper triangular matrices T_j of size n_b ×n_b ($j = 1, ..., b-1$) and an upper trapezoidal n_b×$\min(m,n) - (b-1)n_b$ matrix T_b' ($j = b$) whose upper square part denoted with T_b satisfying

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = \prod_{j=1}^{b} (I - V_j T_j V_j^T)$$

such that v_i is the ith column of V, τ_i is the ith element of `[diag(T_1); diag(T_2); …; diag(T_b)]`, and $(V_1 \; V_2 \; ... \; V_b)$ is the left `m×min(m, n)` block of V. When constructed using `qr`, the block size is given by $n_b = \min(m, n, 36)$.

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors`, as in the `QR` type, is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format such that `V = I + tril(F.factors, -1)`.
- `T` is a n_b-by-$\min(m, n)$ matrix as described above. The subdiagonal elements for each triangular matrix T_j are ignored.

> **Note**
>
> This format should not to be confused with the older *WY* representation [Bischof1987].

`LinearAlgebra.QRPivoted` — Type

```
QRPivoted <: Factorization
```

A QR matrix factorization with column pivoting in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q`, `R`, and `p`.

The object has three fields:

- `factors` is an `m×n` matrix.

  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.
- `jpvt` is an integer vector of length `n` corresponding to the permutation P.

`LinearAlgebra.qr` — Function

```
qr(A, pivot=Val(false); blocksize) -> F
```

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

$$A = QR$$

The returned object `F` stores the factorization in a packed format:

- if `pivot == Val(true)` then `F` is a `QRPivoted` object,
- otherwise if the element type of `A` is a BLAS type (`Float32`, `Float64`, `ComplexF32` or `ComplexF64`), then `F` is a `QRCompactWY` object,
- otherwise `F` is a `QR` object.

The individual components of the decomposition `F` can be retrieved via property accessors:

- `F.Q`: the orthogonal/unitary matrix `Q`
- `F.R`: the upper triangular matrix `R`
- `F.p`: the permutation vector of the pivot (`QRPivoted` only)
- `F.P`: the permutation matrix of the pivot (`QRPivoted` only)

Iterating the decomposition produces the components `Q`, `R`, and if extant `p`.

The following functions are available for the `QR` objects: `inv`, `size`, and `\`. When `A` is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned. When `A` is not full rank, factorization with (column) pivoting is required to obtain a minimum norm solution.

Multiplication with respect to either full/square or non-full/square `Q` is allowed, i.e. both `F.Q*F.R` and `F.Q*A` are supported. A `Q` matrix can be converted into a regular matrix with `Matrix`. This operation returns the "thin" Q factor, i.e., if `A` is `m×n` with `m>=n`, then `Matrix(F.Q)` yields an `m×n` matrix with orthonormal columns. To retrieve the "full" Q factor, an `m×m` orthogonal matrix, use `F.Q*Matrix(I,m,m)`. If `m<=n`, then `Matrix(F.Q)` yields an `m×m` orthogonal matrix.

The block size for QR decomposition can be specified by keyword argument `blocksize :: Integer` when `pivot == Val(false)` and `A isa StridedMatrix{<:BlasFloat}`. It is ignored when `blocksize > minimum(size(A))`. See `QRCompactWY`.

> **Julia 1.4**
>
> The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Array{Float64,2}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qr(A)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
3×3 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.6   0.0   0.8
 -0.8   0.0  -0.6
  0.0  -1.0   0.0
R factor:
2×2 Array{Float64,2}:
 -5.0  10.0
  0.0  -1.0

julia> F.Q * F.R == A
true
```

> **Note**
>
> `qr` returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the `Q` and `R` matrices can be stored compactly rather as two separate dense matrices.

`LinearAlgebra.qr!` — Function

```
qr!(A, pivot=Val(false); blocksize)
```

`qr!` is the same as `qr` when `A` is a subtype of `StridedMatrix`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> a = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> qr!(a)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
2×2 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.316228  -0.948683
 -0.948683   0.316228
R factor:
2×2 Array{Float64,2}:
 -3.16228  -4.42719
  0.0      -0.632456

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> qr!(a)
ERROR: InexactError: Int64(-3.1622776601683795)
Stacktrace:
[...]
```

`LinearAlgebra.LQ` — Type

```
LQ <: Factorization
```

Matrix factorization type of the `LQ` factorization of a matrix `A`. The `LQ` decomposition is the `QR` decomposition of `transpose(A)`. This is the return type of `lq`, the corresponding matrix factorization function.

If `S::LQ` is the factorization object, the lower triangular component can be obtained via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64,Array{Float64,2}} with factors L and Q:
[-8.60233 0.0; 4.41741 -0.697486]
[-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq` — Function

```
lq(A) -> S::LQ
```

Compute the LQ decomposition of `A`. The decomposition's lower triangular component can be obtained from the `LQ` object `S` via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

The LQ decomposition is the QR decomposition of `transpose(A)`, and it is useful in order to compute the minimum-norm solution `lq(A) \ b` to an underdetermined system of equations (`A` has more columns than rows, but has full row rank).

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64,Array{Float64,2}} with factors L and Q:
[-8.60233 0.0; 4.41741 -0.697486]
[-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq!` — Function

```
lq!(A) -> LQ
```

Compute the `LQ` factorization of `A`, using the input matrix as a workspace. See also `lq`.

`LinearAlgebra.BunchKaufman` — Type

```
BunchKaufman <: Factorization
```

Matrix factorization type of the Bunch-Kaufman factorization of a symmetric or Hermitian matrix `A` as `P'UDU'P` or `P'LDL'P`, depending on whether the upper (the default) or the lower triangle is stored in `A`. If `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`, respectively. This is the return type of `bunchkaufman`, the corresponding matrix factorization function.

If `S::BunchKaufman` is the factorization object, the components can be obtained via `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Examples

```
julia> A = [1 2; 2 3]
2×2 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ⋅   1.0
permutation:
2-element Array{Int64,1}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64,Array{Float64,2}}
```

```
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ⋅
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

`LinearAlgebra.bunchkaufman` — Function

```
bunchkaufman(A, rook::Bool=false; check = true) -> S::BunchKaufman
```

Compute the Bunch-Kaufman [Bunch1977] factorization of a symmetric or Hermitian matrix `A` as `P'*U*D*U'*P` or `P'*L*D*L'*P`, depending on which triangle is stored in `A`, and return a `BunchKaufman` object. Note that if `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

If `rook` is `true`, rook pivoting is used. If `rook` is false, rook pivoting is not used.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issymmetric`, `ishermitian`, `getindex`.

Examples

```
julia> A = [1 2; 2 3]
2×2 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
```

```
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ⋅   1.0
permutation:
2-element Array{Int64,1}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ⋅
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

`LinearAlgebra.bunchkaufman!` — Function

```
bunchkaufman!(A, rook::Bool=false; check = true) -> BunchKaufman
```

`bunchkaufman!` is the same as `bunchkaufman`, but saves space by overwriting the input `A`, instead of creating a copy.

`LinearAlgebra.Eigen` — Type

```
Eigen <: Factorization
```

Matrix factorization type of the eigenvalue/spectral decomposition of a square matrix `A`. This is the return type of `eigen`, the corresponding matrix factorization function.

If `F::Eigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
vectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
```

```
true
```

`LinearAlgebra.GeneralizedEigen` — Type

```
GeneralizedEigen <: Factorization
```

Matrix factorization type of the generalized eigenvalue/spectral decomposition of `A` and `B`. This is the return type of `eigen`, the corresponding matrix factorization function, when called with two matrix arguments.

If `F::GeneralizedEigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> F = eigen(A, B)
GeneralizedEigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Ar
values:
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
vectors:
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> F.values
2-element Array{Complex{Float64},1}:
```

```
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

`LinearAlgebra.eigvals` — Function

```
eigvals(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Return the eigenvalues of `A`.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The `permute`, `scale`, and `sortby` keywords are the same as for `eigen!`.

Examples

```
julia> diag_matrix = [1 0; 0 4]
2×2 Array{Int64,2}:
 1  0
 0  4

julia> eigvals(diag_matrix)
2-element Array{Float64,1}:
 1.0
 4.0
```

For a scalar input, `eigvals` will return a scalar.

Example

```
julia> eigvals(-2)
-2
```

```
eigvals(A, B) -> values
```

Computes the generalized eigenvalues of `A` and `B`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> va
```

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a `UnitRange` `irange` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ⋅
```

```
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A, 2:2)
1-element Array{Float64,1}:
 0.9999999999999996

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> v
```

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ⋅
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A, -1, 2)
1-element Array{Float64,1}:
 1.0000000000000009

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

`LinearAlgebra.eigvals!` — Function

```
eigvals!(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

> **Note**
>
> The input matrix `A` will not contain its eigenvalues after `eigvals!` is called on it - `A` is used as a workspace.

Examples

```
julia> A = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> eigvals!(A)
2-element Array{Float64,1}:
 -0.3722813232690143
  5.372281323269014

julia> A
2×2 Array{Float64,2}:
 -0.372281  -1.0
  0.0        5.37228
```

```
eigvals!(A, B; sortby) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A` (and `B`), instead of creating copies.

> **Note**
>
> The input matrices `A` and `B` will not contain their eigenvalues after `eigvals!` is called. They are used as workspaces.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
```

```
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> eigvals!(A, B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> A
2×2 Array{Float64,2}:
 -0.0  -1.0
  1.0  -0.0

julia> B
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> v
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `irange` is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) ->
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `vl` is the lower bound of the interval to search for eigenvalues, and `vu` is the upper bound.

`LinearAlgebra.eigmax` — Function

```
eigmax(A; permute::Bool=true, scale::Bool=true)
```

Return the largest eigenvalue of `A`. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of `A` are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

`LinearAlgebra.eigmin` — Function

```
eigmin(A; permute::Bool=true, scale::Bool=true)
```

Return the smallest eigenvalue of `A`. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of `A` are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
```

```
-1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmin(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

`LinearAlgebra.eigvecs` — Function

```
eigvecs(A::SymTridiagonal[, eigvals]) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

If the optional vector of eigenvalues `eigvals` is specified, `eigvecs` returns the specific corresponding eigenvectors.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ⋅
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259

julia> eigvecs(A)
3×3 Array{Float64,2}:
  0.418304  -0.83205      0.364299
 -0.656749  -7.39009e-16  0.754109
  0.627457   0.5547       0.546448
```

```
julia> eigvecs(A, [1.])
3×1 Array{Float64,2}:
  0.8320502943378438
  4.263514128092366e-17
 -0.5547001962252291
```

```
eigvecs(A; permute::Bool=true, scale::Bool=true, `sortby`) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.) The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
eigvecs(A, B) -> Matrix
```

Return a matrix `M` whose columns are the generalized eigenvectors of `A` and `B`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
```

```
-1.0+0.0im  -1.0-0.0im
```

`LinearAlgebra.eigen` — Function

```
eigen(A; permute::Bool=true, scale::Bool=true, sortby) -> Eigen
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the eigenvalues in an arbitrary order. Some special matrix types (e.g. `Diagonal` or `SymTridiagonal`) may implement their own sorting convention and not accept a `sortby` keyword.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
vectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
```

```
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A, B) -> GeneralizedEigen
```

Computes the generalized eigenvalue decomposition of `A` and `B`, returning a `GeneralizedEigen` factorization object `F` which contains the generalized eigenvalues in `F.values` and the generalized eigenvectors in the columns of the matrix `F.vectors`. (The `k`th generalized eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Any keyword arguments passed to `eigen` are passed through to the lower-level `eigen!` function.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> F = eigen(A, B);

julia> F.values
2-element Array{Complex{Float64},1}:
```

```
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> Eige
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

The `UnitRange` `irange` specifies indices of the sorted eigenvalues to search for.

> **Note**
>
> If `irange` is not `1:n`, where `n` is the dimension of `A`, then the returned factorization will be a *truncated* factorization.

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> Eig
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

`vl` is the lower bound of the window of eigenvalues to search for, and `vu` is the upper bound.

> **Note**
>
> If [`vl`, `vu`] does not contain all eigenvalues of `A`, then the returned factorization will be a *truncated* factorization.

`LinearAlgebra.eigen!` — Function

```
eigen!(A, [B]; permute, scale, sortby)
```

Same as `eigen`, but saves space by overwriting the input `A` (and `B`), instead of creating a copy.

`LinearAlgebra.Hessenberg` — Type

```
Hessenberg <: Factorization
```

A `Hessenberg` object represents the Hessenberg factorization `QHQ'` of a square matrix, or a shift `Q(H+μI)Q'` thereof, which is produced by the `hessenberg` function.

`LinearAlgebra.hessenberg` — Function

```
hessenberg(A) -> Hessenberg
```

Compute the Hessenberg decomposition of `A` and return a `Hessenberg` object. If `F` is the factorization object, the unitary matrix can be accessed with `F.Q` (of type `LinearAlgebra.HessenbergQ`) and the Hessenberg matrix with `F.H` (of type `UpperHessenberg`), either of which may be converted to a regular matrix with `Matrix(F.H)` or `Matrix(F.Q)`.

If `A` is `Hermitian` or real-`Symmetric`, then the Hessenberg decomposition produces a real-symmetric tridiagonal matrix and `F.H` is of type `SymTridiagonal`.

Note that the shifted factorization `A+μI = Q (H+μI) Q'` can be constructed efficiently by `F + μ*I` using the `UniformScaling` object `I`, which creates a new `Hessenberg` object with shared storage and a modified shift. The shift of a given `F` is obtained by `F.μ`. This is useful because multiple shifted solves `(F + μ*I) \ b` (for different `μ` and/or `b`) can be performed efficiently once `F` is created.

Iterating the decomposition produces the factors `F.Q, F.H, F.μ`.

Examples

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessenberg(A)
Hessenberg{Float64,UpperHessenberg{Float64,Array{Float64,2}},Array{Float64,2},A
Q factor:
3×3 LinearAlgebra.HessenbergQ{Float64,Array{Float64,2},Array{Float64,1},false}:
 1.0   0.0        0.0
 0.0  -0.707107  -0.707107
 0.0  -0.707107   0.707107
H factor:
3×3 UpperHessenberg{Float64,Array{Float64,2}}:
  4.0      -11.3137       -1.41421
 -5.65685    5.0           2.0
   ⋅        -8.88178e-16   1.0

julia> F.Q * F.H * F.Q'
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> q, h = F; # destructuring via iteration

julia> q == F.Q && h == F.H
true
```

`LinearAlgebra.hessenberg!` — Function

```
hessenberg!(A) -> Hessenberg
```

`hessenberg!` is the same as `hessenberg`, but saves space by overwriting the input `A`, instead of creating a copy.

`LinearAlgebra.Schur` — Type

```
Schur <: Factorization
```

Matrix factorization type of the Schur factorization of a matrix `A`. This is the return type of `schur(_)`, the corresponding matrix factorization function.

If `F::Schur` is the factorization object, the (quasi) triangular Schur factor can be obtained via either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors via `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0
```

```
julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

`LinearAlgebra.GeneralizedSchur` — Type

```
GeneralizedSchur <: Factorization
```

Matrix factorization type of the generalized Schur factorization of two matrices `A` and `B`. This is the return type of `schur(_, _)`, the corresponding matrix factorization function.

If `F::GeneralizedSchur` is the factorization object, the (quasi) triangular Schur factors can be obtained via `F.S` and `F.T`, the left unitary/orthogonal Schur vectors via `F.left` or `F.Q`, and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur` — Function

```
schur(A::StridedMatrix) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the `Schur` object `F` with either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors can be obtained with `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0
```

```
julia> F = schur(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

```
schur(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices `A` and `B`. The (quasi) triangular Schur factors can be obtained from the `Schur` object `F` with `F.S` and `F.T`, the left unitary/orthogonal Schur vectors can be obtained with `F.left` or `F.Q` and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur!` — Function

```
schur!(A::StridedMatrix) -> F::Schur
```

Same as `schur` but uses the input argument `A` as workspace.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur!(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> A
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
```

```
schur!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as `schur` but uses the input matrices `A` and `B` as workspace.

`LinearAlgebra.ordschur` — Function

```
ordschur(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Reorders the Schur factorization `F` of a matrix `A = Z*T*Z'` according to the logical array `select` returning the reordered factorization `F` object. The selected eigenvalues appear in the leading diagonal of `F.Schur` and the corresponding leading columns of `F.vectors` form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::Gener
```

Reorders the Generalized Schur factorization `F` of a matrix pair `(A, B) = (Q*S*Z', Q*T*Z')` according to the logical array `select` and returns a GeneralizedSchur object `F`. The selected eigenvalues appear in the leading diagonal of both `F.S` and `F.T`, and the left and right orthogonal/unitary Schur vectors are also reordered such that `(A, B) = F.Q*(F.S, F.T)*F.Z'` still holds and the generalized eigenvalues of `A` and `B` can still be obtained with `F.α./F.β`.

`LinearAlgebra.ordschur!` — Function

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as `ordschur` but overwrites the factorization `F`.

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::Gene
```

Same as `ordschur` but overwrites the factorization `F`.

`LinearAlgebra.SVD` — Type

```
SVD <: Factorization
```

Matrix factorization type of the singular value decomposition (SVD) of a matrix `A`. This is the return type of `svd(_)`, the corresponding matrix factorization function.

If `F::SVD` is the factorization object, `U`, `S`, `V` and `Vt` can be obtained via `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
```

```
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> F = svd(A)
SVD{Float64,Float64,Array{Float64,2}}
U factor:
4×4 Array{Float64,2}:
 0.0  1.0  0.0   0.0
 1.0  0.0  0.0   0.0
 0.0  0.0  0.0  -1.0
 0.0  0.0  1.0   0.0
singular values:
4-element Array{Float64,1}:
 3.0
 2.23606797749979
 2.0
 0.0
Vt factor:
4×5 Array{Float64,2}:
 -0.0       0.0  1.0  -0.0  0.0
  0.447214  0.0  0.0   0.0  0.894427
 -0.0       1.0  0.0  -0.0  0.0
  0.0       0.0  0.0   1.0  0.0

julia> F.U * Diagonal(F.S) * F.Vt
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> u, s, v = F; # destructuring via iteration

julia> u == F.U && s == F.S && v == F.V
true
```

`LinearAlgebra.GeneralizedSVD` — Type

```
GeneralizedSVD <: Factorization
```

Matrix factorization type of the generalized singular value decomposition (SVD) of two matrices `A` and `B`, such that `A = F.U*F.D1*F.R0*F.Q'` and `B = F.V*F.D2*F.R0*F.Q'`. This is the return type of `svd(_, _)`, the corresponding matrix factorization function.

For an M-by-N matrix `A` and P-by-N matrix `B`,

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

`K+L` is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The entries of `F.D1` and `F.D2` are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> F = svd(A, B)
GeneralizedSVD{Float64,Array{Float64,2}}
U factor:
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
V factor:
2×2 Array{Float64,2}:
 -0.0  -1.0
```

```
  1.0   0.0
Q factor:
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
D1 factor:
2×2 SparseArrays.SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.707107
  [2, 2]  =  0.707107
D2 factor:
2×2 SparseArrays.SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.707107
  [2, 2]  =  0.707107
R0 factor:
2×2 Array{Float64,2}:
 1.41421   0.0
 0.0      -1.41421

julia> F.U*F.D1*F.R0*F.Q'
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0
```

`LinearAlgebra.svd` — Function

```
svd(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

Compute the singular value decomposition (SVD) of `A` and return an `SVD` object.

`U`, `S`, `V` and `Vt` can be obtained from the factorization `F` with `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

If `full = false` (default), a "thin" SVD is returned. For a $M \times N$ matrix `A`, in the full factorization `U` is `M \times M` and `V` is `N \times N`, while in the thin factorization `U` is `M \times`

`K` and `V` is `N \times K`, where `K = \min(M,N)` is the number of singular values.

If `alg = DivideAndConquer()` a divide-and-conquer algorithm is used to calculate the SVD. Another (typically slower but more accurate) option is `alg = QRIteration()`.

> **Julia 1.3**
>
> The `alg` keyword argument requires Julia 1.3 or later.

Examples

```
julia> A = rand(4,3);

julia> F = svd(A); # Store the Factorization Object

julia> A ≈ F.U * Diagonal(F.S) * F.Vt
true

julia> U, S, V = F; # destructuring via iteration

julia> A ≈ U * Diagonal(S) * V'
true

julia> Uonly, = svd(A); # Store U only

julia> Uonly == U
true
```

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` factorization object `F` such that `[A;B] = [F.U * F.D1; F.V * F.D2] * F.R0 * F.Q'`

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block

triangular,

`K+L` is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The generalized SVD is used in applications such as when one wants to compare how much belongs to `A` vs. how much belongs to `B`, as in human vs yeast genome, or signal vs noise, or between clusters vs within clusters. (See Edelman and Wang for discussion: https://arxiv.org/abs/1901.00485)

It decomposes `[A; B]` into `[UC; VS]H`, where `[UC; VS]` is a natural orthogonal basis for the column space of `[A; B]`, and `H = RQ'` is a natural non-orthogonal basis for the rowspace of `[A;B]`, where the top rows are most closely attributed to the `A` matrix, and the bottom to the `B` matrix. The multi-cosine/sine matrices `C` and `S` provide a multi-measure of how much `A` vs how much `B`, and `U` and `V` provide directions in which these are measured.

Examples

```
julia> A = randn(3,2); B=randn(4,2);

julia> F = svd(A, B);

julia> U,V,Q,C,S,R = F;

julia> H = R*Q';

julia> [A; B] ≈ [U*C; V*S]*H
true

julia> [A; B] ≈ [F.U*F.D1; F.V*F.D2]*F.R0*F.Q'
true

julia> Uonly, = svd(A,B);

julia> U == Uonly
true
```

`LinearAlgebra.svd!` — Function

```
svd!(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

`svd!` is the same as `svd`, but saves space by overwriting the input `A`, instead of creating a copy. See documentation of `svd` for details. ```

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as `svd`, but modifies the arguments `A` and `B` in-place, instead of making copies. See documentation of `svd` for details. ```

`LinearAlgebra.svdvals` — Function

```
svdvals(A)
```

Return the singular values of `A` in descending order.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> svdvals(A)
4-element Array{Float64,1}:
 3.0
 2.23606797749979
 2.0
 0.0
```

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of `A` and `B`. See also `svd`.

Examples

```
julia> A = [1. 0.; 0. -1.]
```

```
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> svdvals(A, B)
2-element Array{Float64,1}:
 1.0
 1.0
```

`LinearAlgebra.svdvals!` — Function

```
svdvals!(A)
```

Return the singular values of `A`, saving space by overwriting the input. See also `svdvals` and `svd`. ```

```
svdvals!(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of `A` and `B`, saving space by overwriting `A` and `B`. See also `svd` and `svdvals`. ```

`LinearAlgebra.Givens` — Type

```
LinearAlgebra.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields `c` and `s` represent the cosine and sine of the rotation angle, respectively. The `Givens` type supports left multiplication `G*A` and conjugated transpose right multiplication `A*G'`. The type doesn't have a `size` and can therefore be multiplied with matrices of arbitrary size as long as `i2<=size(A,2)` for `G*A` or `i2<=size(A,1)` for `A*G'`.

See also: `givens`

`LinearAlgebra.givens` — Function

```
givens(f::T, g::T, i1::Integer, i2::Integer) where {T} -> (G::Givens, r::T)
```

Computes the Givens rotation `G` and scalar `r` such that for any vector `x` where

```
x[i1] = f
x[i2] = g
```

the result of the multiplication

```
y = G*x
```

has the property that

```
y[i1] = r
y[i2] = 0
```

See also: `LinearAlgebra.Givens`

```
givens(A::AbstractArray, i1::Integer, i2::Integer, j::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*A
```

has the property that

```
B[i1,j] = r
B[i2,j] = 0
```

See also: `LinearAlgebra.Givens`

```
givens(x::AbstractVector, i1::Integer, i2::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*x
```

has the property that

```
B[i1] = r
B[i2] = 0
```

See also: `LinearAlgebra.Givens`

`LinearAlgebra.triu` — Function

```
triu(M)
```

Upper triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0
 0.0  0.0  0.0  1.0
```

```
triu(M, k::Integer)
```

Returns the upper triangle of `M` starting from the `k`th superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
```

```
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a,3)
4×4 Array{Float64,2}:
 0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> triu(a,-3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

`LinearAlgebra.triu!` — Function

```
triu!(M)
```

Upper triangle of a matrix, overwriting `M` in the process. See also `triu`.

```
triu!(M, k::Integer)
```

Return the upper triangle of `M` starting from the `k`th superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

```
julia> triu!(M, 1)
5×5 Array{Int64,2}:
 0  2  3  4  5
 0  0  3  4  5
 0  0  0  4  5
 0  0  0  0  5
 0  0  0  0  0
```

`LinearAlgebra.tril` — Function

```
tril(M)
```

Lower triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a)
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 1.0  1.0  0.0  0.0
 1.0  1.0  1.0  0.0
 1.0  1.0  1.0  1.0
```

```
tril(M, k::Integer)
```

Returns the lower triangle of `M` starting from the `k`th superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,-3)
4×4 Array{Float64,2}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

`LinearAlgebra.tril!` — Function

```
tril!(M)
```

Lower triangle of a matrix, overwriting `M` in the process. See also `tril`.

```
tril!(M, k::Integer)
```

Return the lower triangle of `M` starting from the `k`th superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> tril!(M, 2)
5×5 Array{Int64,2}:
 1  2  3  0  0
```

```
 1  2  3  4  0
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

`LinearAlgebra.diagind` — Function

```
diagind(M, k::Integer=0)
```

An `AbstractRange` giving the indices of the `k`th diagonal of the matrix `M`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diagind(A,-1)
2:4:6
```

`LinearAlgebra.diag` — Function

```
diag(M, k::Integer=0)
```

The `k`th diagonal of a matrix, as a vector.

See also: `diagm`

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
```

```
2-element Array{Int64,1}:
 2
 6
```

`LinearAlgebra.diagm` — Function

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
diagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a matrix from `Pair`s of diagonals and vectors. Vector `kv.second` will be placed on the `kv.first` diagonal. By default the matrix is square and its size is inferred from `kv`, but a non-square size `m`×`n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments.

`diagm` constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see `Diagonal`, `Bidiagonal` `Tridiagonal` and `SymTridiagonal`.

Examples

```
julia> diagm(1 => [1,2,3])
4×4 Array{Int64,2}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], -1 => [4,5])
4×4 Array{Int64,2}:
 0  1  0  0
 4  0  2  0
 0  5  0  3
 0  0  0  0
```

```
diagm(v::AbstractVector)
diagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a matrix with elements of the vector as diagonal elements. By default (if `size=nothing`), the matrix is square and its size is given by `length(v)`, but a non-square size `m`×`n` can be specified by passing `m,n` as the first arguments.

Examples

```
julia> diagm([1,2,3])
3×3 Array{Int64,2}:
 1  0  0
 0  2  0
 0  0  3
```

`LinearAlgebra.rank` — Function

```
rank(A::AbstractMatrix; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
rank(A::AbstractMatrix, rtol::Real)
```

Compute the rank of a matrix by counting how many singular values of `A` have magnitude greater than `max(atol, rtol*σ₁)` where `σ₁` is `A`'s largest singular value. `atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `A`, and `ϵ` is the `eps` of the element type of `A`.

> **Julia 1.1**
>
> The `atol` and `rtol` keyword arguments requires at least Julia 1.1. In Julia 1.0 `rtol` is available as a positional argument, but this will be deprecated in Julia 2.0.

Examples

```
julia> rank(Matrix(I, 3, 3))
3

julia> rank(diagm(0 => [1, 0, 2]))
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.1)
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.00001)
3

julia> rank(diagm(0 => [1, 0.001, 2]), atol=1.5)
1
```

`LinearAlgebra.norm` — Function

```
norm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the `p`-norm (defaulting to `p=2`) as if `A` were a vector of the corresponding length.

The `p`-norm is defined as

$$\|A\|_p = \left(\sum_{i=1}^n |a_i|^p \right)^{1/p}$$

with a_i the entries of A, $|a_i|$ the `norm` of a_i, and n the length of A. Since the `p`-norm is computed using the `norm`s of the entries of `A`, the `p`-norm of a vector of vectors is not compatible with the interpretation of it as a block vector in general if `p != 2`.

`p` can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs.(A)`, whereas `norm(A, -Inf)` returns the smallest. If `A` is a matrix and `p=2`, then this is equivalent to the Frobenius norm.

The second argument `p` is not necessarily a part of the interface for `norm`, i.e. a custom type may only implement `norm(A)` without second argument.

Use `opnorm` to compute the operator norm of a matrix.

Examples

```
julia> v = [3, -2, 6]
3-element Array{Int64,1}:
  3
 -2
  6

julia> norm(v)
7.0

julia> norm(v, 1)
11.0

julia> norm(v, Inf)
6.0

julia> norm([1 2 3; 4 5 6; 7 8 9])
16.881943016134134

julia> norm([1 2 3 4 5 6 7 8 9])
16.881943016134134

julia> norm(1:9)
16.881943016134134

julia> norm(hcat(v,v), 1) == norm(vcat(v,v), 1) != norm([v,v], 1)
true

julia> norm(hcat(v,v), 2) == norm(vcat(v,v), 2) == norm([v,v], 2)
true

julia> norm(hcat(v,v), Inf) == norm(vcat(v,v), Inf) != norm([v,v], Inf)
true
```

```
norm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$.

Examples

```
julia> norm(2, 1)
```

```
2.0

julia> norm(-2, 1)
2.0

julia> norm(2, 2)
2.0

julia> norm(-2, 2)
2.0

julia> norm(2, Inf)
2.0

julia> norm(-2, Inf)
2.0
```

`LinearAlgebra.opnorm` — Function

```
opnorm(A::AbstractMatrix, p::Real=2)
```

Compute the operator norm (or matrix norm) induced by the vector `p`-norm, where valid values of `p` are `1`, `2`, or `Inf`. (Note that for sparse matrices, `p=2` is currently not implemented.) Use `norm` to compute the Frobenius norm.

When `p=1`, the operator norm is the maximum absolute column sum of `A`:

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When `p=2`, the operator norm is the spectral norm, equal to the largest singular value of `A`.

When `p=Inf`, the operator norm is the maximum absolute row sum of `A`:

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$$

Examples

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Array{Int64,2}:
 1  -2  -3
 2   3  -1

julia> opnorm(A, Inf)
6.0

julia> opnorm(A, 1)
5.0
```

```
opnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to `norm`.

```
opnorm(A::Adjoint{<:Any,<:AbstractVector}, q::Real=2)
opnorm(A::Transpose{<:Any,<:AbstractVector}, q::Real=2)
```

For Adjoint/Transpose-wrapped vectors, return the operator q-norm of `A`, which is equivalent to the `p`-norm with value `p = q/(q-1)`. They coincide at `p = q = 2`. Use `norm` to compute the `p` norm of `A` as a vector.

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the dot product, and the result is consistent with the operator `p`-norm of a `1 × n` matrix.

Examples

```
julia> v = [1; im];

julia> vc = v';

julia> opnorm(vc, 1)
1.0

julia> norm(vc, 1)
2.0

julia> norm(v, 1)
2.0
```

```
julia> opnorm(vc, 2)
1.4142135623730951

julia> norm(vc, 2)
1.4142135623730951

julia> norm(v, 2)
1.4142135623730951

julia> opnorm(vc, Inf)
2.0

julia> norm(vc, Inf)
1.0

julia> norm(v, Inf)
1.0
```

`LinearAlgebra.normalize!` — Function

```
normalize!(a::AbstractArray, p::Real=2)
```

Normalize the array `a` in-place so that its `p`-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize` and `norm`.

`LinearAlgebra.normalize` — Function

```
normalize(a::AbstractArray, p::Real=2)
```

Normalize the array `a` so that its `p`-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize!` and `norm`.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Array{Float64,1}:
```

```
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Array{Float64,1}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714

julia> norm(c, 1)
1.0

julia> a = [1 2 4 ; 1 2 4]
2×3 Array{Int64,2}:
 1  2  4
 1  2  4

julia> norm(a)
6.48074069840786

julia> normalize(a)
2×3 Array{Float64,2}:
 0.154303  0.308607  0.617213
 0.154303  0.308607  0.617213
```

`LinearAlgebra.cond` — Function

```
cond(M, p::Real=2)
```

Condition number of the matrix `M`, computed using the operator `p`-norm. Valid values for `p` are `1`, `2` (default), or `Inf`.

`LinearAlgebra.condskeel` — Function

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M, p) = \left\Vert \left\vert M \right\vert \left\vert M^{-1} \right\vert \right\Vert_p$$
$$\kappa_S(M, x, p) = \frac{\left\Vert \left\vert M \right\vert \left\vert M^{-1} \right\vert \left\vert x \right\vert \right\Vert_p}{\left\Vert x \right\Vert_p}$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator `p`-norm. $\left\vert M \right\vert$ denotes the matrix of (entry wise) absolute values of M; $\left\vert M \right\vert_{ij} = \left\vert M_{ij} \right\vert$. Valid values for `p` are `1`, `2` and `Inf` (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

`LinearAlgebra.tr` — Function

```
tr(M)
```

Matrix trace. Sums the diagonal elements of `M`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> tr(A)
5
```

`LinearAlgebra.det` — Function

```
det(M)
```

Matrix determinant.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
```

```
 1  0
 2  2

julia> det(M)
2.0
```

`LinearAlgebra.logdet` — Function

```
logdet(M)
```

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> logdet(M)
0.6931471805599453

julia> logdet(Matrix(I, 3, 3))
0.0
```

`LinearAlgebra.logabsdet` — Function

```
logabsdet(M)
```

Log of absolute value of matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

Examples

```
julia> A = [-1. 0.; 0. 1.]
2×2 Array{Float64,2}:
 -1.0  0.0
```

```
  0.0  1.0

julia> det(A)
-1.0

julia> logabsdet(A)
(0.0, -1.0)

julia> B = [2. 0.; 0. 1.]
2×2 Array{Float64,2}:
 2.0  0.0
 0.0  1.0

julia> det(B)
2.0

julia> logabsdet(B)
(0.6931471805599453, 1.0)
```

`Base.inv` — Method

```
inv(M)
```

Matrix inverse. Computes matrix `N` such that `M * N = I`, where `I` is the identity matrix. Computed by solving the left-division `N = M \ I`.

Examples

```
julia> M = [2 5; 1 3]
2×2 Array{Int64,2}:
 2  5
 1  3

julia> N = inv(M)
2×2 Array{Float64,2}:
  3.0  -5.0
 -1.0   2.0

julia> M*N == N*M == Matrix(I, 2, 2)
true
```

LinearAlgebra.pinv — Function

```
pinv(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
pinv(M, rtol::Real) = pinv(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes the Moore-Penrose pseudoinverse.

For matrices `M` with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values greater than `max(atol, rtol*σ₁)` where `σ₁` is the largest singular value of `M`.

The optimal choice of absolute (`atol`) and relative tolerance (`rtol`) varies both with the value of `M` and the intended application of the pseudoinverse. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the `eps` of the element type of `M`.

For inverting dense ill-conditioned matrices in a least-squares sense, `rtol = sqrt(eps(real(float(one(eltype(M))))))` is recommended.

For more information, see [issue8859], [B96], [S84], [KY88].

Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Array{Float64,2}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
2×2 Array{Float64,2}:
  1.47287   -1.00775
 -0.930233   1.16279

julia> M * N
2×2 Array{Float64,2}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

LinearAlgebra.nullspace — Function

```
nullspace(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
```

```
nullspace(M, rtol::Real) = nullspace(M; rtol=rtol) # to be deprecated in Julia
```

Computes a basis for the nullspace of `M` by including the singular vectors of `M` whose singular values have magnitudes greater than `max(atol, rtol*σ₁)`, where `σ₁` is `M`'s largest singular value.

By default, the relative tolerance `rtol` is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the `eps` of the element type of `M`.

Examples

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0

julia> nullspace(M, rtol=3)
3×3 Array{Float64,2}:
 0.0  1.0  0.0
 1.0  0.0  0.0
 0.0  0.0  1.0

julia> nullspace(M, atol=0.95)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

`Base.kron` — Function

```
kron(A, B)
```

Kronecker tensor product of two vectors or two matrices.

For real vectors `v` and `w`, the Kronecker product is related to the outer product by `kron(v,w) == vec(w * transpose(v))` or `w * transpose(v) == reshape(kron(v,w), (length(w), length(v)))`. Note how the ordering of `v` and `w` differs on the left and right of these expressions (due to column-major storage). For complex vectors, the outer product `w * v'` also differs by conjugation of `v`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [im 1; 1 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Array{Complex{Int64},2}:
 0+1im  1+0im  0+2im  2+0im
 1+0im  0-1im  2+0im  0-2im
 0+3im  3+0im  0+4im  4+0im
 3+0im  0-3im  4+0im  0-4im

julia> v = [1, 2]; w = [3, 4, 5];

julia> w*transpose(v)
3×2 Array{Int64,2}:
 3   6
 4   8
 5  10

julia> reshape(kron(v,w), (length(w), length(v)))
3×2 Array{Int64,2}:
 3   6
 4   8
 5  10
```

`Base.exp` — Method

```
exp(A::AbstractMatrix)
```

Compute the matrix exponential of `A`, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian `A`, an eigendecomposition (`eigen`) is used, otherwise the scaling and squaring algorithm (see [H05]) is chosen.

Examples

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> exp(A)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828
```

`Base.:^` — Method

```
^(A::AbstractMatrix, p::Number)
```

Matrix power, equivalent to $\exp(p \log(A))$

Examples

```
julia> [1 2; 0 3]^3
2×2 Array{Int64,2}:
 1  26
 0  27
```

`Base.:^` — Method

```
^(b::Number, A::AbstractMatrix)
```

Matrix exponential, equivalent to $\exp(\log(b)A)$.

> **Julia 1.1**
>
> Support for raising `Irrational` numbers (like e) to a matrix was added in Julia 1.1.

Examples

```
julia> 2^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.0  6.0
 0.0  8.0

julia> ℯ^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.71828  17.3673
 0.0      20.0855
```

`Base.log` — Method

```
log(A{T}::StridedMatrix{T})
```

If `A` has no negative real eigenvalue, compute the principal matrix logarithm of `A`, i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If `A` has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used, if `A` is triangular an improved version of the inverse scaling and squaring method is employed (see [AH12] and [AHR13]). For general matrices, the complex Schur form (`schur`) is computed and the triangular algorithm is used on the triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Array{Float64,2}:
```

```
 2.71828  0.0
 0.0      2.71828

julia> log(A)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

`Base.sqrt` — Method

```
sqrt(A::AbstractMatrix)
```

If `A` has no negative real eigenvalues, compute the principal matrix square root of `A`, that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If `A` is real-symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the square root. For such matrices, eigenvalues λ that appear to be slightly negative due to roundoff errors are treated as if they were zero More precisely, matrices with all eigenvalues `≥ -rtol*(max |λ|)` are treated as semidefinite (yielding a Hermitian square root), with negative eigenvalues taken to be zero. `rtol` is a keyword argument to `sqrt` (in the Hermitian/real-symmetric case only) that defaults to machine precision scaled by `size(A,1)`.

Otherwise, the square root is determined by means of the Björck-Hammarling method [BH83], which computes the complex Schur form (`schur`) and then the complex square root of the triangular factor.

Examples

```
julia> A = [4 0; 0 4]
2×2 Array{Int64,2}:
 4  0
 0  4

julia> sqrt(A)
2×2 Array{Float64,2}:
 2.0  0.0
 0.0  2.0
```

Base.cos — Method

```
cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the cosine. Otherwise, the cosine is determined by calling `exp`.

Examples

```
julia> cos(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Base.sin — Method

```
sin(A::AbstractMatrix)
```

Compute the matrix sine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the sine. Otherwise, the sine is determined by calling `exp`.

Examples

```
julia> sin(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649
```

Base.Math.sincos — Method

```
sincos(A::AbstractMatrix)
```

Compute the matrix sine and cosine of a square matrix `A`.

Examples

```
julia> S, C = sincos(fill(1.0, (2,2)));

julia> S
2×2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649

julia> C
2×2 Array{Float64,2}:
  0.291927  -0.708073
 -0.708073   0.291927
```

`Base.tan` — Method

```
tan(A::AbstractMatrix)
```

Compute the matrix tangent of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the tangent. Otherwise, the tangent is determined by calling `exp`.

Examples

```
julia> tan(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
 -1.09252  -1.09252
 -1.09252  -1.09252
```

`Base.Math.sec` — Method

```
sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix `A`.

`Base.Math.csc` — Method

```
csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix `A`.

`Base.Math.cot` — Method

```
cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix `A`.

`Base.cosh` — Method

```
cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix `A`.

`Base.sinh` — Method

```
sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix `A`.

`Base.tanh` — Method

```
tanh(A::AbstractMatrix)
```

Compute the matrix hyperbolic tangent of a square matrix `A`.

`Base.Math.sech` — Method

```
sech(A::AbstractMatrix)
```

Compute the matrix hyperbolic secant of square matrix `A`.

`Base.Math.csch` — Method

```
csch(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosecant of square matrix `A`.

`Base.Math.coth` — Method

```
coth(A::AbstractMatrix)
```

Compute the matrix hyperbolic cotangent of square matrix `A`.

`Base.acos` — Method

```
acos(A::AbstractMatrix)
```

Compute the inverse matrix cosine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse cosine. Otherwise, the inverse cosine is determined by using `log` and `sqrt`. For the theory and logarithmic formulas used to compute this function, see [AH16_1].

Examples

```
julia> acos(cos([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5-8.32667e-17im  0.1+0.0im
 -0.2+2.63678e-16im  0.3-3.46945e-16im
```

`Base.asin` — Method

```
asin(A::AbstractMatrix)
```

Compute the inverse matrix sine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse sine. Otherwise, the inverse sine is determined by using `log` and `sqrt`. For the theory and logarithmic formulas used to compute this function, see [AH16_2].

Examples

```
julia> asin(sin([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5-4.16334e-17im  0.1-5.55112e-17im
 -0.2+9.71445e-17im  0.3-1.249e-16im
```

`Base.atan` — Method

```
atan(A::AbstractMatrix)
```

Compute the inverse matrix tangent of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse tangent. Otherwise, the inverse tangent is determined by using `log`. For the theory and logarithmic formulas used to compute this function, see [AH16_3].

Examples

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5+1.38778e-17im  0.1-2.77556e-17im
 -0.2+6.93889e-17im  0.3-4.16334e-17im
```

`Base.Math.asec` — Method

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of `A`.

`Base.Math.acsc` — Method

```
acsc(A::AbstractMatrix)
```

Compute the inverse matrix cosecant of `A`.

`Base.Math.acot` — Method

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of `A`.

`Base.acosh` — Method

```
acosh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix cosine of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [AH16_4].

`Base.asinh` — Method

```
asinh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix sine of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [AH16_5].

`Base.atanh` — Method

```
atanh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix tangent of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [AH16_6].

`Base.Math.asech` — Method

```
asech(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic secant of `A`.

`Base.Math.acsch` — Method

```
acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of `A`.

`Base.Math.acoth` — Method

```
acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of `A`.

`LinearAlgebra.lyap` — Function

```
lyap(A, C)
```

Computes the solution `X` to the continuous Lyapunov equation `AX + XA' + C = 0`, where no eigenvalue of `A` has a zero real part and no two eigenvalues are negative complex conjugates of each other.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> X = lyap(A, B)
2×2 Array{Float64,2}:
  0.5  -0.5
 -0.5   0.25

julia> A*X + X*A' + B
2×2 Array{Float64,2}:
 0.0          6.66134e-16
 6.66134e-16  8.88178e-16
```

`LinearAlgebra.sylvester` — Function

```
sylvester(A, B, C)
```

Computes the solution `X` to the Sylvester equation `AX + XB + C = 0`, where `A`, `B` and `C` have compatible dimensions and `A` and `-B` have no eigenvalues with equal real part.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> C = [1. 2.; -2. 1]
2×2 Array{Float64,2}:
```

```
  1.0  2.0
 -2.0  1.0

julia> X = sylvester(A, B, C)
2×2 Array{Float64,2}:
 -4.46667   1.93333
  3.73333  -1.8

julia> A*X + X*B + C
2×2 Array{Float64,2}:
  2.66454e-15  1.77636e-15
 -3.77476e-15  4.44089e-16
```

`LinearAlgebra.issuccess` — Function

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

```
julia> F = cholesky([1 0; 0 1]);

julia> LinearAlgebra.issuccess(F)
true

julia> F = lu([1 0; 0 0]; check = false);

julia> LinearAlgebra.issuccess(F)
false
```

`LinearAlgebra.issymmetric` — Function

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
```

```
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> issymmetric(b)
false
```

`LinearAlgebra.isposdef` — Function

```
isposdef(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of `A`. See also `isposdef!`

Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1   2
 2  50

julia> isposdef(A)
true
```

`LinearAlgebra.isposdef!` — Function

```
isposdef!(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of `A`, overwriting `A` in the process. See also `isposdef`.

Examples

```
julia> A = [1. 2.; 2. 50.];

julia> isposdef!(A)
true

julia> A
2×2 Array{Float64,2}:
 1.0  2.0
 2.0  6.78233
```

`LinearAlgebra.istril` — Function

```
istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether `A` is lower triangular starting from the `k`th superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istril(a)
false

julia> istril(a, 1)
true

julia> b = [1 0; -im -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+0im
 0-1im  -1+0im

julia> istril(b)
true

julia> istril(b, -1)
false
```

`LinearAlgebra.istriu` — Function

```
istriu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether `A` is upper triangular starting from the `k`th superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1
```

```
julia> istriu(a)
false

julia> istriu(a, -1)
true

julia> b = [1 im; 0 -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+1im
 0+0im  -1+0im

julia> istriu(b)
true

julia> istriu(b, 1)
false
```

`LinearAlgebra.isdiag` — Function

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true
```

`LinearAlgebra.ishermitian` — Function

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
```

```
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

`Base.transpose` — Function

```
transpose(A)
```

Lazy transpose. Mutating the returned object should appropriately mutate `A`. Often, but not always, yields `Transpose(A)`, where `Transpose` is a lazy transpose wrapper. Note that this operation is recursive.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

`LinearAlgebra.transpose!` — Function

```
transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> transpose!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3+2im  8+7im
 9+2im  4+6im

julia> A
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im
```

`LinearAlgebra.Transpose` — Type

```
Transpose
```

Lazy wrapper type for a transpose view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`, but also some `Factorization`, for instance. Usually, the `Transpose` constructor should not be called directly, use `transpose` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
```

```
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

`Base.adjoint` — Function

```
A'
adjoint(A)
```

Lazy adjoint (conjugate transposition). Note that `adjoint` is applied recursively to elements.

For number types, `adjoint` returns the complex conjugate, and therefore it is equivalent to the identity function for real numbers.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im

julia> x = [3, 4im]
2-element Array{Complex{Int64},1}:
 3 + 0im
 0 + 4im

julia> x'x
25 + 0im
```

`LinearAlgebra.adjoint!` — Function

```
adjoint!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> adjoint!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3-2im  8-7im
 9-2im  4-6im

julia> A
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im
```

`LinearAlgebra.Adjoint` — Type

```
Adjoint
```

Lazy wrapper type for an adjoint view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`, but also some `Factorization`, for instance. Usually, the `Adjoint` constructor should not be called directly, use `adjoint` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im
```

`Base.copy` — Method

```
copy(A::Transpose)
copy(A::Adjoint)
```

Eagerly evaluate the lazy matrix transpose/adjoint. Note that the transposition is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [1 2im; -3im 4]
2×2 Array{Complex{Int64},2}:
 1+0im  0+2im
 0-3im  4+0im

julia> T = transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Array{Complex{Int64},2}:
 1+0im  0-3im
 0+2im  4+0im
```

`LinearAlgebra.stride1` — Function

```
stride1(A) -> Int
```

Return the distance between successive array elements in dimension 1 in units of element size.

Examples

```
julia> A = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
julia> LinearAlgebra.stride1(A)
1

julia> B = view(A, 2:2:4)
2-element view(::Array{Int64,1}, 2:2:4) with eltype Int64:
 2
 4

julia> LinearAlgebra.stride1(B)
2
```

`LinearAlgebra.checksquare` — Function

```
LinearAlgebra.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

```
julia> A = fill(1, (4,4)); B = fill(1, (5,5));

julia> LinearAlgebra.checksquare(A, B)
2-element Array{Int64,1}:
 4
 5
```

`LinearAlgebra.peakflops` — Function

```
LinearAlgebra.peakflops(n::Integer=2000; parallel::Bool=false)
```

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies a matrix of size `n x n`, where `n = 2000`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

Julia 1.1

This function requires at least Julia 1.1. In Julia 1.0 it is available from the standard library `InteractiveUtils`.

Low-level matrix operations

In many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with `!` below (e.g. `mul!`) according to the usual Julia convention.

`LinearAlgebra.mul!` — Function

```
mul!(Y, A, B) -> Y
```

Calculates the matrix-matrix or matrix-vector product AB and stores the result in `Y`, overwriting the existing value of `Y`. Note that `Y` must not be aliased with either `A` or `B`.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; Y = similar(B); mul!(Y, A, B

julia> Y
2×2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

Implementation

For custom matrix and vector types, it is recommended to implement 5-argument `mul!` rather than implementing 3-argument `mul!` directly if possible.

```
mul!(C, A, B, α, β) -> C
```

Combined inplace matrix-matrix or matrix-vector multiply-add $AB\alpha + C\beta$. The result is stored in `C` by overwriting it. Note that `C` must not be aliased with either `A` or `B`.

> **Julia 1.3**
>
> Five-argument `mul!` requires at least Julia 1.3.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; C=[1.0 2.0; 3.0 4.0];

julia> mul!(C, A, B, 100.0, 10.0) === C
true

julia> C
2×2 Array{Float64,2}:
 310.0  320.0
 730.0  740.0
```

`LinearAlgebra.lmul!` — Function

```
lmul!(a::Number, B::AbstractArray)
```

Scale an array `B` by a scalar `a` overwriting `B` in-place. Use `rmul!` to multiply scalar from right. The scaling operation respects the semantics of the multiplication `*` between `a` and an element of `B`. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

> **Julia 1.1**
>
> Prior to Julia 1.1, `NaN` and `±Inf` entries in `B` were treated inconsistently.

Examples

```
julia> B = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> lmul!(2, B)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> lmul!(0.0, [Inf])
1-element Array{Float64,1}:
 NaN
```

```
lmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting `B`, and return the result. Here, `A` must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> B = [0 1; 1 0];

julia> A = LinearAlgebra.UpperTriangular([1 2; 0 3]);

julia> LinearAlgebra.lmul!(A, B);

julia> B
2×2 Array{Int64,2}:
 2  1
 3  0

julia> B = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> lmul!(F.Q, B)
2×2 Array{Float64,2}:
 3.0  4.0
 1.0  2.0
```

`LinearAlgebra.rmul!` — Function

```
rmul!(A::AbstractArray, b::Number)
```

Scale an array `A` by a scalar `b` overwriting `A` in-place. Use `lmul!` to multiply scalar from left. The scaling operation respects the semantics of the multiplication `*` between an element of `A` and `b`. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

Julia 1.1

Prior to Julia 1.1, `NaN` and `±Inf` entries in `A` were treated inconsistently.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rmul!(A, 2)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> rmul!([NaN], 0.0)
1-element Array{Float64,1}:
 NaN
```

```
rmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting `A`, and return the result. Here, `B` must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> A = [0 1; 1 0];
```

```
julia> B = LinearAlgebra.UpperTriangular([1 2; 0 3]);

julia> LinearAlgebra.rmul!(A, B);

julia> A
2×2 Array{Int64,2}:
 0  3
 1  2

julia> A = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> rmul!(A, F.Q)
2×2 Array{Float64,2}:
 2.0  1.0
 4.0  3.0
```

`LinearAlgebra.ldiv!` — Function

```
ldiv!(Y, A, B) -> Y
```

Compute `A \ B` in-place and store the result in `Y`, returning the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = zero(X);

julia> ldiv!(Y, qr(A), X);

julia> Y
```

```
3-element Array{Float64,1}:
  0.7128099173553719
 -0.051652892561983674
  0.10020661157024757

julia> A\X
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
ldiv!(A, B)
```

Compute `A \ B` in-place and overwriting `B` to store the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = copy(X);

julia> ldiv!(qr(A), X);

julia> X
3-element Array{Float64,1}:
  0.7128099173553719
 -0.051652892561983674
  0.10020661157024757

julia> A\Y
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
ldiv!(a::Number, B::AbstractArray)
```

Divide each entry in an array `B` by a scalar `a` overwriting `B` in-place. Use `rdiv!` to divide scalar from right.

Examples

```
julia> B = [1.0 2.0; 3.0 4.0]
```

```
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> ldiv!(2.0, B)
2×2 Array{Float64,2}:
 0.5  1.0
 1.5  2.0
```

`LinearAlgebra.rdiv!` — Function

```
rdiv!(A, B)
```

Compute `A / B` in-place and overwriting `A` to store the result.

The argument `B` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `rdiv!` usually also require fine-grained control over the factorization of `B`.

```
rdiv!(A::AbstractArray, b::Number)
```

Divide each entry in an array `A` by a scalar `b` overwriting `A` in-place. Use `ldiv!` to divide scalar from left.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> rdiv!(A, 2.0)
2×2 Array{Float64,2}:
 0.5  1.0
 1.5  2.0
```

BLAS functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the LAPACK library, which in turn is built on top of basic linear-algebra building-blocks known as the BLAS. There are highly optimized implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

`LinearAlgebra.BLAS` provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in `'!'`. Usually, a BLAS function has four methods defined, for `Float64`, `Float32`, `ComplexF64`, and `ComplexF32` arrays.

BLAS character arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (`trans`), which triangle of a matrix to reference (`uplo` or `ul`), whether the diagonal of a triangular matrix can be assumed to be all ones (`dA`) or which side of a matrix multiplication the input argument belongs on (`side`). The possibilities are:

Multiplication order

`side`	Meaning
`'L'`	The argument goes on the *left* side of a matrix-matrix operation.
`'R'`	The argument goes on the *right* side of a matrix-matrix operation.

Triangle referencing

`uplo`/`ul`	Meaning
`'U'`	Only the *upper* triangle of the matrix will be used.
`'L'`	Only the *lower* triangle of the matrix will be used.

Transposition operation

`trans`/`tX`	Meaning
`'N'`	The input matrix `X` is not transposed or conjugated.
`'T'`	The input matrix `X` will be transposed.

`'C'`	The input matrix `X` will be conjugated and transposed.

Unit diagonal

diag/dX	Meaning
`'N'`	The diagonal values of the matrix `X` will be read.
`'U'`	The diagonal of the matrix `X` is assumed to be all ones.

`LinearAlgebra.BLAS` — Module

Interface to BLAS subroutines.

`LinearAlgebra.BLAS.dot` — Function

```
dot(n, X, incx, Y, incy)
```

Dot product of two vectors consisting of `n` elements of array `X` with stride `incx` and `n` elements of array `Y` with stride `incy`.

Examples

```
julia> BLAS.dot(10, fill(1.0, 10), 1, fill(1.0, 20), 2)
10.0
```

`LinearAlgebra.BLAS.dotu` — Function

```
dotu(n, X, incx, Y, incy)
```

Dot function for two complex vectors consisting of `n` elements of array `X` with stride `incx` and `n` elements of array `Y` with stride `incy`.

Examples

```
julia> BLAS.dotu(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
```

```
-10.0 + 10.0im
```

`LinearAlgebra.BLAS.dotc` — Function

```
dotc(n, X, incx, U, incy)
```

Dot function for two complex vectors, consisting of `n` elements of array `X` with stride `incx` and `n` elements of array `U` with stride `incy`, conjugating the first vector.

Examples

```
julia> BLAS.dotc(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
10.0 - 10.0im
```

`LinearAlgebra.BLAS.blascopy!` — Function

```
blascopy!(n, X, incx, Y, incy)
```

Copy `n` elements of array `X` with stride `incx` to array `Y` with stride `incy`. Returns `Y`.

`LinearAlgebra.BLAS.nrm2` — Function

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

Examples

```
julia> BLAS.nrm2(4, fill(1.0, 8), 2)
2.0

julia> BLAS.nrm2(1, fill(1.0, 8), 2)
1.0
```

`LinearAlgebra.BLAS.asum` — Function

```
asum(n, X, incx)
```

Sum of the absolute values of the first `n` elements of array `X` with stride `incx`.

Examples

```
julia> BLAS.asum(5, fill(1.0im, 10), 2)
5.0

julia> BLAS.asum(2, fill(1.0im, 10), 5)
2.0
```

`LinearAlgebra.axpy!` — Function

```
axpy!(a, X, Y)
```

Overwrite `Y` with `X*a + Y`, where `a` is a scalar. Return `Y`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> BLAS.axpy!(2, x, y)
3-element Array{Int64,1}:
  6
  9
 12
```

`LinearAlgebra.axpby!` — Function

```
axpby!(a, X, b, Y)
```

Overwrite `Y` with `X*a + Y*b`, where `a` and `b` are scalars. Return `Y`.

Examples

```
julia> x = [1., 2, 3];

julia> y = [4., 5, 6];

julia> BLAS.axpby!(2., x, 3., y)
3-element Array{Float64,1}:
 14.0
 19.0
 24.0
```

`LinearAlgebra.BLAS.scal!` — Function

```
scal!(n, a, X, incx)
```

Overwrite `X` with `a*X` for the first `n` elements of array `X` with stride `incx`. Returns `X`.

`LinearAlgebra.BLAS.scal` — Function

```
scal(n, a, X, incx)
```

Return `X` scaled by `a` for the first `n` elements of array `X` with stride `incx`.

`LinearAlgebra.BLAS.iamax` — Function

```
iamax(n, dx, incx)
iamax(dx)
```

Find the index of the element of `dx` with the maximum absolute value. `n` is the length of `dx`, and `incx` is the stride. If `n` and `incx` are not provided, they assume default values of `n=length(dx)` and `incx=stride1(dx)`.

`LinearAlgebra.BLAS.ger!` — Function

```
ger!(alpha, x, y, A)
```

Rank-1 update of the matrix `A` with vectors `x` and `y` as `alpha*x*y' + A`.

`LinearAlgebra.BLAS.syr!` — Function

```
syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix `A` with vector `x` as `alpha*x*transpose(x) + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`LinearAlgebra.BLAS.syrk!` — Function

```
syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix `C` as `alpha*A*transpose(A) + beta*C` or `alpha*transpose(A)*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Returns `C`.

`LinearAlgebra.BLAS.syrk` — Function

```
syrk(uplo, trans, alpha, A)
```

Returns either the upper triangle or the lower triangle of `A`, according to `uplo`, of `alpha*A*transpose(A)` or `alpha*transpose(A)*A`, according to `trans`.

`LinearAlgebra.BLAS.syr2k!` — Function

```
syr2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the symmetric matrix `C` as `alpha*A*transpose(B) + alpha*B*transpose(A) + beta*C` or `alpha*transpose(A)*B + alpha*transpose(B)*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Returns `C`.

`LinearAlgebra.BLAS.syr2k` — Function

```
syr2k(uplo, trans, alpha, A, B)
```

Returns the `uplo` triangle of `alpha*A*transpose(B) + alpha*B*transpose(A)` or `alpha*transpose(A)*B + alpha*transpose(B)*A`, according to `trans`.

```
syr2k(uplo, trans, A, B)
```

Returns the `uplo` triangle of `A*transpose(B) + B*transpose(A)` or `transpose(A)*B + transpose(B)*A`, according to `trans`.

`LinearAlgebra.BLAS.her!` — Function

```
her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix `A` with vector `x` as `alpha*x*x' + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`LinearAlgebra.BLAS.herk!` — Function

```
herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix `C` as `alpha*A*A' + beta*C` or `alpha*A'*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is updated. Returns `C`.

`LinearAlgebra.BLAS.herk` — Function

```
herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the `uplo` triangle of `alpha*A*A'` or `alpha*A'*A`, according to `trans`.

`LinearAlgebra.BLAS.her2k!` — Function

```
her2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the Hermitian matrix `C` as `alpha*A*B' + alpha*B*A' + beta*C` or `alpha*A'*B + alpha*B'*A + beta*C` according to `trans`. The scalar `beta` has to be real. Only the `uplo` triangle of `C` is used. Returns `C`.

`LinearAlgebra.BLAS.her2k` — Function

```
her2k(uplo, trans, alpha, A, B)
```

Returns the `uplo` triangle of `alpha*A*B' + alpha*B*A'` or `alpha*A'*B + alpha*B'*A`, according to `trans`.

```
her2k(uplo, trans, A, B)
```

Returns the `uplo` triangle of `A*B' + B*A'` or `A'*B + B'*A`, according to `trans`.

`LinearAlgebra.BLAS.gbmv!` — Function

```
gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector `y` as `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.gbmv` — Function

```
gbmv(trans, m, kl, ku, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'*x` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals, and `alpha` is a scalar.

`LinearAlgebra.BLAS.sbmv!` — Function

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector `y` as `alpha*A*x + beta*y` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. The storage layout for `A` is described the reference BLAS module, level-2 BLAS at http://www.netlib.org/lapack/explore-html/. Only the `uplo` triangle of `A` is used.

Return the updated `y`.

`LinearAlgebra.BLAS.sbmv` — Method

```
sbmv(uplo, k, alpha, A, x)
```

Return `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.sbmv` — Method

```
sbmv(uplo, k, A, x)
```

Return `A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.gemm!` — Function

```
gemm!(tA, tB, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` and `tB`. Return the updated `C`.

`LinearAlgebra.BLAS.gemm` — Method

```
gemm(tA, tB, alpha, A, B)
```

Return `alpha*A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemm` — Method

```
gemm(tA, tB, A, B)
```

Return `A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemv!` — Function

```
gemv!(tA, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y` or `alpha*A'x + beta*y` according to `tA`. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.gemv` — Method

```
gemv(tA, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'x` according to `tA`. `alpha` is a scalar.

`LinearAlgebra.BLAS.gemv` — Method

```
gemv(tA, A, x)
```

Return `A*x` or `A'x` according to `tA`.

`LinearAlgebra.BLAS.symm!` — Function

```
symm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Return the updated `C`.

`LinearAlgebra.BLAS.symm` — Method

```
symm(side, ul, alpha, A, B)
```

Return `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.symm` — Method

```
symm(side, ul, A, B)
```

Return `A*B` or `B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.symv!` — Function

```
symv!(ul, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.symv` — Method

```
symv(ul, alpha, A, x)
```

Return `alpha*A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` is a scalar.

`LinearAlgebra.BLAS.symv` — Method

```
symv(ul, A, x)
```

Return `A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.hemm!` — Function

```
hemm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used. Return the updated `C`.

`LinearAlgebra.BLAS.hemm` — Method

```
hemm(side, ul, alpha, A, B)
```

Return `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.hemm` — Method

```
hemm(side, ul, A, B)
```

Return `A*B` or `B*A` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.hemv!` — Function

```
hemv!(ul, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.hemv` — Method

```
hemv(ul, alpha, A, x)
```

Return `alpha*A*x`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used. `alpha` is a scalar.

`LinearAlgebra.BLAS.hemv` — Method

```
hemv(ul, A, x)
```

Return `A*x`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.trmm!` — Function

```
trmm!(side, ul, tA, dA, alpha, A, B)
```

Update `B` as `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

`LinearAlgebra.BLAS.trmm` — Function

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Returns `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

`LinearAlgebra.BLAS.trsm!` — Function

```
trsm!(side, ul, tA, dA, alpha, A, B)
```

Overwrite `B` with the solution to `A*X = alpha*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

`LinearAlgebra.BLAS.trsm` — Function

```
trsm(side, ul, tA, dA, alpha, A, B)
```

Return the solution to `A*X = alpha*B` or one of the other three variants determined by determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

`LinearAlgebra.BLAS.trmv!` — Function

```
trmv!(ul, tA, dA, A, b)
```

Return `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on `b`.

`LinearAlgebra.BLAS.trmv` — Function

```
trmv(ul, tA, dA, A, b)
```

Return `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

`LinearAlgebra.BLAS.trsv!` — Function

```
trsv!(ul, tA, dA, A, b)
```

Overwrite `b` with the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones. Return the updated `b`.

`LinearAlgebra.BLAS.trsv` — Function

```
trsv(ul, tA, dA, A, b)
```

Return the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones.

`LinearAlgebra.BLAS.set_num_threads` — Function

```
set_num_threads(n)
```

Set the number of threads the BLAS library should use.

LAPACK functions

`LinearAlgebra.LAPACK` provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in `'!'`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `ComplexF64` and `ComplexF32` arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

`LinearAlgebra.LAPACK` — Module

Interfaces to LAPACK subroutines.

`LinearAlgebra.LAPACK.gbtrf!` — Function

```
gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix `AB`. `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix `AB`. Returns the LU factorization in-place and `ipiv`, the vector of pivots used.

`LinearAlgebra.LAPACK.gbtrs!` — Function

```
gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation `AB * X = B`. `trans` determines the orientation of `AB`. It may be `N` (no transpose), `T` (transpose), or `C` (conjugate transpose). `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix `AB`. `ipiv` is the vector of pivots returned from `gbtrf!`. Returns the vector or matrix `X`, overwriting `B` in-place.

`LinearAlgebra.LAPACK.gebal!` — Function

```
gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix `A` before computing its eigensystem or Schur factorization. `job` can be one of `N` (`A` will not be permuted or scaled), `P` (`A` will only be permuted), `S` (`A` will only be scaled), or `B` (`A` will be both permuted and scaled). Modifies `A` in-place and returns `ilo`, `ihi`, and `scale`. If permuting was turned on, `A[i,j] = 0` if `j > i` and `1 < j < ilo` or `j > ihi`. `scale` contains information about the scaling/permutations performed.

`LinearAlgebra.LAPACK.gebak!` — Function

```
gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors `V` of a matrix balanced using `gebal!` to the unscaled/unpermuted eigenvectors of the original matrix. Modifies `V` in-place. `side` can be `L` (left eigenvectors are transformed) or `R` (right eigenvectors are transformed).

`LinearAlgebra.LAPACK.gebrd!` — Function

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce `A` in-place to bidiagonal form `A = QBP'`. Returns `A`, containing the bidiagonal matrix `B`; `d`, containing the diagonal elements of `B`; `e`, containing the off-diagonal elements of `B`; `tauq`, containing the elementary reflectors representing `Q`; and `taup`, containing the elementary reflectors representing `P`.

`LinearAlgebra.LAPACK.gelqf!` — Function

```
gelqf!(A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
gelqf!(A) -> (A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqlf!` — Function

```
geqlf!(A, tau)
```

Compute the `QL` factorization of `A`, `A = QL`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
geqlf!(A) -> (A, tau)
```

Compute the `QL` factorization of `A`, `A = QL`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqrf!` — Function

```
geqrf!(A, tau)
```

Compute the `QR` factorization of `A`, `A = QR`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
geqrf!(A) -> (A, tau)
```

Compute the `QR` factorization of `A`, `A = QR`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqp3!` — Function

```
geqp3!(A, [jpvt, tau]) -> (A, tau, jpvt)
```

Compute the pivoted `QR` factorization of `A`, `AP = QR` using BLAS level 3. `P` is a pivoting matrix, represented by `jpvt`. `tau` stores the elementary reflectors. The arguments `jpvt` and `tau` are optional and allow for passing preallocated arrays. When passed, `jpvt` must have length greater than or equal to `n` if `A` is an `(m x n)` matrix and `tau` must have length greater than or equal to the smallest dimension of `A`.

`A`, `jpvt`, and `tau` are modified in-place.

`LinearAlgebra.LAPACK.gerqf!` — Function

```
gerqf!(A, tau)
```

Compute the `RQ` factorization of `A`, `A = RQ`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
gerqf!(A) -> (A, tau)
```

Compute the `RQ` factorization of `A`, `A = RQ`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqrt!` — Function

```
geqrt!(A, T)
```

Compute the blocked `QR` factorization of `A`, `A = QR`. `T` contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of `T` sets the block size and it must be between 1 and `n`. The second dimension of `T` must equal the smallest dimension of `A`.

Returns `A` and `T` modified in-place.

```
geqrt!(A, nb) -> (A, T)
```

Compute the blocked `QR` factorization of `A`, `A = QR`. `nb` sets the block size and it must be between 1 and `n`, the second dimension of `A`.

Returns `A`, modified in-place, and `T`, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqrt3!` — Function

```
geqrt3!(A, T)
```

Recursively computes the blocked `QR` factorization of `A`, `A = QR`. `T` contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of `T` sets the block size and it must be between 1 and `n`. The second dimension of `T` must equal the smallest dimension of `A`.

Returns `A` and `T` modified in-place.

```
geqrt3!(A) -> (A, T)
```

Recursively computes the blocked `QR` factorization of `A`, `A = QR`.

Returns `A`, modified in-place, and `T`, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.getrf!` — Function

```
getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted `LU` factorization of `A`, `A = LU`.

Returns `A`, modified in-place, `ipiv`, the pivoting information, and an `info` code which indicates success (`info = 0`), a singular value in `U` (`info = i`, in which case `U[i,i]` is singular), or an error code (`info < 0`).

`LinearAlgebra.LAPACK.tzrzf!` — Function

```
tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix `A` to upper triangular form in-place. Returns `A` and `tau`, the scalar parameters for the elementary reflectors of the transformation.

`LinearAlgebra.LAPACK.ormrz!` — Function

```
ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix `C` by `Q` from the transformation supplied by `tzrzf!`. Depending on `side` or `trans` the multiplication can be left-sided (`side = L, Q*C`) or right-sided (`side = R, C*Q`) and `Q` can be unmodified (`trans = N`), transposed (`trans = T`), or conjugate transposed (`trans = C`). Returns matrix `C` which is modified in-place with the result of the multiplication.

`LinearAlgebra.LAPACK.gels!` — Function

```
gels!(trans, A, B) -> (F, B, ssr)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` using a QR or LQ factorization. Modifies the matrix/vector `B` in place with the solution. `A` is overwritten with its `QR` or `LQ` factorization. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose). `gels!` searches for the minimum norm/least squares solution. `A` may be under or over determined. The solution is returned in `B`.

`LinearAlgebra.LAPACK.gesv!` — Function

```
gesv!(A, B) -> (B, A, ipiv)
```

Solves the linear equation `A * X = B` where `A` is a square matrix using the `LU` factorization of `A`. `A` is overwritten with its `LU` factorization and `B` is overwritten with the solution `X`. `ipiv` contains the pivoting information for the `LU` factorization of `A`.

`LinearAlgebra.LAPACK.getrs!` — Function

```
getrs!(trans, A, ipiv, B)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` for square `A`. Modifies the matrix/vector `B` in place with the solution. `A` is the `LU` factorization from `getrf!`, with `ipiv` the pivoting information. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose).

`LinearAlgebra.LAPACK.getri!` — Function

```
getri!(A, ipiv)
```

Computes the inverse of `A`, using its `LU` factorization found by `getrf!`. `ipiv` is the pivot information output and `A` contains the `LU` factorization of `getrf!`. `A` is overwritten with its inverse.

`LinearAlgebra.LAPACK.gesvx!` — Function

```
gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed, R, C, B, rcond,
```

Solves the linear equation `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) using the `LU` factorization of `A`. `fact` may be `E`, in which case `A` will be equilibrated and copied to `AF`; `F`, in which case `AF` and `ipiv` from a previous `LU` factorization are inputs; or `N`, in which case `A` will be copied to `AF` and then factored. If `fact = F`, `equed` may be `N`, meaning `A` has not been equilibrated; `R`, meaning `A` was multiplied by `Diagonal(R)` from the left; `C`, meaning `A` was multiplied by `Diagonal(C)` from the right; or `B`, meaning `A` was multiplied by `Diagonal(R)` from the left and `Diagonal(C)` from the right. If `fact = F` and `equed = R` or `B` the elements of `R` must all be positive. If `fact = F` and `equed = C` or `B` the elements of `C` must all be positive.

Returns the solution `X`; `equed`, which is an output if `fact` is not `N`, and describes the equilibration that was performed; `R`, the row equilibration diagonal; `C`, the column equilibration diagonal; `B`, which may be overwritten with its equilibrated form `Diagonal(R)*B` (if `trans = N` and `equed = R,B`) or `Diagonal(C)*B` (if `trans = T,C` and `equed = C,B`); `rcond`, the reciprocal condition number of `A` after equilbrating; `ferr`, the forward error bound for each solution vector in `X`; `berr`, the forward error bound for each solution vector in `X`; and `work`, the reciprocal pivot growth factor.

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of `gesvx!`.

`LinearAlgebra.LAPACK.gelsd!` — Function

```
gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the `SVD` factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

`LinearAlgebra.LAPACK.gelsy!` — Function

```
gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the full `QR` factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

`LinearAlgebra.LAPACK.gglse!` — Function

```
gglse!(A, c, B, d) -> (X,res)
```

Solves the equation `A * x = c` where `x` is subject to the equality constraint `B * x = d`. Uses the formula `||c - A*x||^2 = 0` to solve. Returns `X` and the residual sum-of-squares.

`LinearAlgebra.LAPACK.geev!` — Function

```
geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of `A`. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. Returns the eigenvalues in `W`, the right eigenvectors in `VR`, and the left eigenvectors in `VL`.

`LinearAlgebra.LAPACK.gesdd!` — Function

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`, using a divide and conquer approach. If `job = A`, all the columns of `U` and the rows of `V'` are computed. If `job = N`, no columns of `U` or rows of `V'` are computed. If `job = O`, `A` is overwritten with the columns of (thin) `U` and the rows of (thin) `V'`. If `job = S`, the columns of (thin) `U` and the rows of (thin) `V'` are computed and returned separately.

`LinearAlgebra.LAPACK.gesvd!` — Function

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`. If `jobu = A`, all the columns of `U` are computed. If `jobvt = A` all the rows of `V'` are computed. If `jobu = N`, no columns of `U` are computed. If `jobvt = N` no rows of `V'` are computed. If `jobu = O`, `A` is overwritten with the columns of (thin) `U`. If `jobvt = O`, `A` is overwritten with the rows of (thin) `V'`. If `jobu = S`, the columns of (thin) `U` are computed and returned separately. If `jobvt = S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `O`.

Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

`LinearAlgebra.LAPACK.ggsvd!` — Function

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of `A` and `B`, `U'*A*Q = D1*R` and `V'*B*Q = D2*R`. `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu = U`, the orthogonal/unitary matrix `U` is computed. If `jobv = V` the orthogonal/unitary matrix `V` is computed. If `jobq = Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv` or `jobq` is `N`, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

`LinearAlgebra.LAPACK.ggsvd3!` — Function

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of `A` and `B`, `U'*A*Q = D1*R` and `V'*B*Q = D2*R`. `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu = U`, the orthogonal/unitary matrix `U` is computed. If `jobv = V` the orthogonal/unitary matrix `V` is computed. If `jobq = Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv`, or `jobq` is `N`, that matrix is not computed. This function requires LAPACK 3.6.0.

`LinearAlgebra.LAPACK.geevx!` — Function

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm
```

Finds the eigensystem of `A` with matrix balancing. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. If `balanc = N`, no balancing is performed. If `balanc = P`, `A` is permuted but not scaled. If `balanc = S`, `A` is scaled but not permuted. If `balanc = B`, `A` is permuted and scaled. If `sense = N`, no reciprocal condition numbers are computed. If `sense = E`, reciprocal condition numbers are computed for the eigenvalues only. If `sense = V`, reciprocal condition numbers are computed for the right eigenvectors only. If `sense = B`, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If `sense = E,B`, the right and left eigenvectors must be computed.

`LinearAlgebra.LAPACK.ggev!` — Function

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of `A` and `B`. If `jobvl = N`, the left eigenvectors aren't computed. If `jobvr = N`, the right eigenvectors aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed.

`LinearAlgebra.LAPACK.gtsv!` — Function

```
gtsv!(dl, d, du, B)
```

Solves the equation `A * X = B` where `A` is a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Overwrites `B` with the solution `X` and returns it.

`LinearAlgebra.LAPACK.gttrf!` — Function

```
gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the `LU` factorization of a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Modifies `dl`, `d`, and `du` in-place and returns them and the second superdiagonal `du2` and the pivoting vector `ipiv`.

`LinearAlgebra.LAPACK.gttrs!` — Function

```
gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) using the `LU` factorization computed by `gttrf!`. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.orglq!` — Function

```
orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `LQ` factorization after calling `gelqf!` on `A`. Uses the output of `gelqf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgqr!` — Function

```
orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QR` factorization after calling `geqrf!` on `A`. Uses the output of `geqrf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgql!` — Function

```
orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QL` factorization after calling `geqlf!` on `A`. Uses the output of `geqlf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgrq!` — Function

```
orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `RQ` factorization after calling `gerqf!` on `A`. Uses the output of `gerqf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.ormlq!` — Function

```
ormlq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `LQ` factorization of `A` computed using `gelqf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormqr!` — Function

```
ormqr!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormql!` — Function

```
ormql!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QL` factorization of `A` computed using `geqlf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormrq!` — Function

```
ormrq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `RQ` factorization of `A` computed using `gerqf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.gemqrt!` — Function

```
gemqrt!(side, trans, V, T, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrt!`. `C` is overwritten.

`LinearAlgebra.LAPACK.posv!` — Function

```
posv!(uplo, A, B) -> (A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix. If `uplo = U` the upper Cholesky decomposition of `A` is computed. If `uplo = L` the lower Cholesky decomposition of `A` is computed. `A` is overwritten by its Cholesky decomposition. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.potrf!` — Function

```
potrf!(uplo, A)
```

Computes the Cholesky (upper if `uplo = U`, lower if `uplo = L`) decomposition of positive-definite matrix `A`. `A` is overwritten and returned with an info code.

`LinearAlgebra.LAPACK.potri!` — Function

```
potri!(uplo, A)
```

Computes the inverse of positive-definite matrix `A` after calling `potrf!` to find its (upper if `uplo = U`, lower if `uplo = L`) Cholesky decomposition.

`A` is overwritten by its inverse and returned.

`LinearAlgebra.LAPACK.potrs!` — Function

```
potrs!(uplo, A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `potrf!`. If `uplo = U` the upper Cholesky decomposition of `A` was computed. If `uplo = L` the lower Cholesky decomposition of `A` was computed. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.pstrf!` — Function

```
pstrf!(uplo, A, tol) -> (A, piv, rank, info)
```

Computes the (upper if `uplo = U`, lower if `uplo = L`) pivoted Cholesky decomposition of positive-definite matrix `A` with a user-set tolerance `tol`. `A` is overwritten by its Cholesky decomposition.

Returns `A`, the pivots `piv`, the rank of `A`, and an `info` code. If `info = 0`, the factorization succeeded. If `info = i > 0`, then `A` is indefinite or rank-deficient.

`LinearAlgebra.LAPACK.ptsv!` — Function

```
ptsv!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A`. `D` is the diagonal of `A` and `E` is the off-diagonal. `B` is overwritten with the solution `X` and returned.

`LinearAlgebra.LAPACK.pttrf!` — Function

```
pttrf!(D, E)
```

Computes the LDLt factorization of a positive-definite tridiagonal matrix with `D` as diagonal and `E` as off-diagonal. `D` and `E` are overwritten and returned.

`LinearAlgebra.LAPACK.pttrs!` — Function

```
pttrs!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A` with diagonal `D` and off-diagonal `E` after computing `A`'s LDLt factorization using `pttrf!`. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.trtri!` — Function

```
trtri!(uplo, diag, A)
```

Finds the inverse of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `A` is overwritten with its inverse.

`LinearAlgebra.LAPACK.trtrs!` — Function

```
trtrs!(uplo, trans, diag, A, B)
```

Solves `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) for (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.trcon!` — Function

```
trcon!(norm, uplo, diag, A)
```

Finds the reciprocal condition number of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. If `norm = I`, the condition number is found in the infinity norm. If `norm = O` or `1`, the condition number is found in the one norm.

`LinearAlgebra.LAPACK.trevc!` — Function

```
trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix `T`. If `side = R`, the right eigenvectors are computed. If `side = L`, the left eigenvectors are computed. If `side = B`, both sets are computed. If `howmny = A`, all eigenvectors are found. If `howmny = B`, all eigenvectors are found and backtransformed using `VL` and `VR`. If `howmny = S`, only the eigenvectors corresponding to the values in `select` are computed.

`LinearAlgebra.LAPACK.trrfs!` — Function

```
trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), `adjoint(A) * X = B` (`trans = C`) for `side = L`, or the equivalent equations a right-handed `side = R` `X * A` after computing `X` using `trtrs!`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, `A` is lower triangular. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `Ferr` and `Berr` are optional inputs. `Ferr` is the forward error and `Berr` is the backward error, each component-wise.

`LinearAlgebra.LAPACK.stev!` — Function

```
stev!(job, dv, ev) -> (dv, Zmat)
```

Computes the eigensystem for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `job = N` only the eigenvalues are found and returned in `dv`. If `job = V` then the eigenvectors are also found and returned in `Zmat`.

`LinearAlgebra.LAPACK.stebz!` — Function

```
stebz!(range, order, vl, vu, il, iu, abstol, dv, ev) -> (dv, iblock, isplit)
```

Computes the eigenvalues for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. If `order = B`, eigvalues are ordered within a block. If `order = E`, they are ordered across all the blocks. `abstol` can be set as a tolerance for convergence.

`LinearAlgebra.LAPACK.stegr!` — Function

```
stegr!(jobz, range, dv, ev, vl, vu, il, iu) -> (w, Z)
```

Computes the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. The eigenvalues are returned in `w` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.stein!` — Function

```
stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

`LinearAlgebra.LAPACK.syconv!` — Function

```
syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

`LinearAlgebra.LAPACK.sysv!` — Function

```
sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.sytrf!` — Function

```
sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.sytri!` — Function

```
sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.sytrs!` — Function

```
sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.hesv!` — Function

```
hesv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.hetrf!` — Function

```
hetrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.hetri!` — Function

```
hetri!(uplo, A, ipiv)
```

Computes the inverse of a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.hetrs!` — Function

```
hetrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.syev!` — Function

```
syev!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

`LinearAlgebra.LAPACK.syevr!` — Function

```
syevr!(jobz, range, uplo, A, vl, vu, il, iu, abstol) -> (W, Z)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.sygvd!` — Function

```
sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)
```

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A` and symmetric positive-definite matrix `B`. If `uplo = U`, the upper triangles of `A` and `B` are used. If `uplo = L`, the lower triangles of `A` and `B` are used. If `itype = 1`, the problem to solve is `A * x = lambda * B * x`. If `itype = 2`, the problem to solve is `A * B * x = lambda * x`. If `itype = 3`, the problem to solve is `B * A * x = lambda * x`.

`LinearAlgebra.LAPACK.bdsqr!` — Function

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product `Q' * C`.

Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

`LinearAlgebra.LAPACK.bdsdc!` — Function

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal using a divide and conqueq method. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. If `compq = N`, only the singular values are found. If `compq = I`, the singular values and vectors are found. If `compq = P`, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in `d`, and if `compq = P`, the compact singular vectors in `iq`.

`LinearAlgebra.LAPACK.gecon!` — Function

```
gecon!(normtype, A, anorm)
```

Finds the reciprocal condition number of matrix `A`. If `normtype = I`, the condition number is found in the infinity norm. If `normtype = O` or `1`, the condition number is found in the one norm. `A` must be the result of `getrf!` and `anorm` is the norm of `A` in the relevant norm.

`LinearAlgebra.LAPACK.gehrd!` — Function

```
gehrd!(ilo, ihi, A) -> (A, tau)
```

Converts a matrix `A` to Hessenberg form. If `A` is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo = 1` and `ihi = size(A,2)`. `tau` contains the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.orghr!` — Function

```
orghr!(ilo, ihi, A, tau)
```

Explicitly finds `Q`, the orthogonal/unitary matrix from `gehrd!`. `ilo`, `ihi`, `A`, and `tau` must correspond to the input/output to `gehrd!`.

`LinearAlgebra.LAPACK.gees!` — Function

```
gees!(jobvs, A) -> (A, vs, w)
```

Computes the eigenvalues (`jobvs = N`) or the eigenvalues and Schur vectors (`jobvs = V`) of matrix `A`. `A` is overwritten by its Schur form.

Returns `A`, `vs` containing the Schur vectors, and `w`, containing the eigenvalues.

`LinearAlgebra.LAPACK.gges!` — Function

```
gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B`.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

`LinearAlgebra.LAPACK.trexc!` — Function

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization of a matrix. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. `ifst` and `ilst` specify the reordering of the vectors.

`LinearAlgebra.LAPACK.trsen!` — Function

```
trsen!(compq, job, select, T, Q) -> (T, Q, w, s, sep)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If `job = N`, no condition numbers are found. If `job = E`, only the condition number for this cluster of eigenvalues is found. If `job = V`, only the condition number for the invariant subspace is found. If `job = B` then the condition numbers for the cluster and subspace are found. If `compq = V` the Schur vectors `Q` are updated. If `compq = N` the Schur vectors are not modified. `select` determines which eigenvalues are in the cluster.

Returns `T`, `Q`, reordered eigenvalues in `w`, the condition number of the cluster of eigenvalues `s`, and the condition number of the invariant subspace `sep`.

`LinearAlgebra.LAPACK.tgsen!` — Function

```
tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. `select` specifies the eigenvalues in each cluster.

`LinearAlgebra.LAPACK.trsyl!` — Function

```
trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation `A * X +/- X * B = scale*C` where `A` and `B` are both quasi-upper triangular. If `transa = N`, `A` is not modified. If `transa = T`, `A` is transposed. If `transa = C`, `A` is conjugate transposed. Similarly for `transb` and `B`. If `isgn = 1`, the equation `A * X + X * B = scale * C` is solved. If `isgn = -1`, the equation `A * X - X * B = scale * C` is solved.

Returns `X` (overwriting `C`) and `scale`.

- Bischof1987 C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009
- Schreiber1989 R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005
- Bunch1977 J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.
- issue8859 Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859
- B96 Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484
- S84 G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030
- KY88 Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585
- H05 Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539
- AH12 Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553
- AHR13 Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991
- BH83 Åke Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X
- AH16_1 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and

Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

- AH16_2 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577
- AH16_3 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577
- AH16_4 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577
- AH16_5 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577
- AH16_6 Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

« Dynamic Linker Logging »