# Sparse Cholesky decomposition (**sksparse.cholmod**)



*New in version 0.1.*

## Overview

This module provides efficient implementations of all the basic linear algebra operations for sparse, symmetric, positive-definite matrices (as, for instance, commonly arise in least squares problems).

Specifically, it exposes most of the capabilities of the CHOLMOD package, including:

- Computation of the Cholesky decomposition $LL' = A$ or $LDL' = A$ (with fill-reducing permutation) for both real and complex sparse matrices $A$, in any format supported by **scipy.sparse**. (However, CSC matrices will be most efficient.)
- A convenient and efficient interface for using this decomposition to solve problems of the form $Ax = b$.
- The ability to perform the costly fill-reduction analysis once, and then re-use it to efficiently decompose many matrices with the same pattern of non-zero entries.
- In-place 'update' and 'downdate' operations, for computing the Cholesky decomposition of a rank-k update of $A$ and of product $AA'$. So, the result is the Cholesky decomposition of $A + CC'$ (or $AA' + CC'$). The last case is useful when the columns of $A$ become available incrementally (e.g., due to memory constraints), or when many matrices with similar but non-identical columns must be factored.
- Convenience functions for computing the (log) determinant of the matrix that has been factored.
- A convenience function for explicitly computing the inverse of the matrix that has been factored (though this is rarely useful).

## Quickstart

If $A$ is a sparse, symmetric, positive-definite matrix, and $b$ is a matrix or vector (either sparse or dense), then the following code solves the equation $Ax = b$:

```
from sksparse.cholmod import cholesky
factor = cholesky(A)
x = factor(b)
```

If we just want to compute its determinant:

```
factor = cholesky(A)
ld = factor.logdet()
```

(This returns the log of the determinant, rather than the determinant itself, to avoid issues with underflow/overflow. See **logdet()**, **log()**.)

If you have a least-squares problem to solve, minimizing $||Mx - b||^2$, and $M$ is a sparse matrix, the solution is $x = (M'M)^{-1}M'b$, which can be efficiently calculated as:

```
from sksparse.cholmod import cholesky_AAt
# Notice that CHOLMOD computes AA' and we want M'M, so we must set A = M'!
factor = cholesky_AAt(M.T)
x = factor(M.T * b)
```



However, you should be aware that for least squares problems, the Cholesky method is usually faster but somewhat less numerically stable than QR- or SVD-based techniques.

# Top-level functions

All usage of this module starts by calling one of four functions, all of which return a **Factor** object, documented below.

Most users will want one of the `cholesky` functions, which perform a fill-reduction analysis and decomposition together:

`sksparse.cholmod.`**`cholesky`**(*A, beta=0, mode="auto", ordering_method="default", use_long=None*)

Computes the fill-reducing Cholesky decomposition of

$$A + \beta I$$

where `A` is a sparse, symmetric, positive-definite matrix, preferably in CSC format, and `beta` is any real scalar (usually 0 or 1). (And $I$ denotes the identity matrix.)

Only the lower triangular part of `A` is used.

`mode` is passed to **`analyze()`**.

`ordering_method` is passed to **`analyze()`**.

`use_long` is passed to **`analyze()`**.

**Returns:** A **`Factor`** object represented the decomposition.

`sksparse.cholmod.`**`cholesky_AAt`**(*A, beta=0, mode="auto", ordering_method="default", use_long=None*)

Computes the fill-reducing Cholesky decomposition of

$$AA' + \beta I$$

where `A` is a sparse matrix, preferably in CSC format, and `beta` is any real scalar (usually 0 or 1). (And $I$ denotes the identity matrix.)

Note that if you are solving a conventional least-squares problem, you will need to transpose your matrix before calling this function, and therefore it will be somewhat more efficient to construct your matrix in CSR format (so that its transpose will be in CSC format).

`mode` is passed to **`analyze_AAt()`**.

`ordering_method` is passed to **`analyze_AAt()`**.

`use_long` is passed to **`analyze_AAt()`**.

**Returns:** A **`Factor`** object represented the decomposition.

However, some users may want to break the fill-reduction analysis and actual decomposition into separate steps, and instead begin with one of the `analyze` functions, which perform only fill-reduction:

`sksparse.cholmod.`**`analyze`**(*A, mode="auto", ordering_method="default", use_long=None*)

Computes the optimal fill-reducing permutation for the symmetric matrix A, but does *not* factor it (i.e., it performs a "symbolic Cholesky decomposition"). This function ignores the actual contents of the matrix A. All it cares about are (1) which entries are non-zero, and (2) whether A has real or complex type.

**Parameters:**
- **A** – The matrix to be analyzed.
- **mode** – Specifies which algorithm should be used to (eventually) compute the Cholesky decomposition – one of "simplicial", "supernodal", or

"auto". See the CHOLMOD documentation for details on how "auto" chooses the algorithm to be used.

- **ordering_method** – Specifies which ordering algorithm should be used to (eventually) order the matrix A – one of "natural", "amd", "metis", "nesdis", "colamd", "default" and "best". "natural" means no permutation. See the CHOLMOD documentation for more details.
- **use_long** – Specifies if the long type (64 bit) or the int type (32 bit) should be used for the indices of the sparse matrices. If use_long is None try to estimate if long type is needed.

**Returns:** A **Factor** object representing the analysis. Many operations on this object will fail, because it does not yet hold a full decomposition. Use **Factor.cholesky_inplace()** (or similar) to actually factor a matrix.

sksparse.cholmod.**analyze_AAt**(*A*, *mode="auto"*, *ordering_method="default"*, *use_long=None*)

Computes the optimal fill-reducing permutation for the symmetric matrix $AA'$, but does *not* factor it (i.e., it performs a "symbolic Cholesky decomposition"). This function ignores the actual contents of the matrix A. All it cares about are (1) which entries are non-zero, and (2) whether A has real or complex type.

**Parameters:**

- **A** – The matrix to be analyzed.
- **mode** – Specifies which algorithm should be used to (eventually) compute the Cholesky decomposition – one of "simplicial", "supernodal", or "auto". See the CHOLMOD documentation for details on how "auto" chooses the algorithm to be used.
- **ordering_method** – Specifies which ordering algorithm should be used to (eventually) order the matrix A – one of "natural", "amd", "metis", "nesdis", "colamd", "default" and "best". "natural" means no permutation. See the CHOLMOD documentation for more details.
- **use_long** – Specifies if the long type (64 bit) or the int type (32 bit) should be used for the indices of the sparse matrices. If use_long is None try to estimate if long type is needed.

**Returns:** A **Factor** object representing the analysis. Many operations on this object will fail, because it does not yet hold a full decomposition. Use **Factor.cholesky_AAt_inplace()** (or similar) to actually factor a matrix.

### Note:

Even if you used **cholesky()** or **cholesky_AAt()**, you can still call **cholesky_inplace()** or **cholesky_AAt_inplace()** on the resulting **Factor** to quickly factor another matrix with the same non-zero pattern as your original matrix.

## **Factor** objects

*class* sksparse.cholmod.**Factor**

A **Factor** object represents the Cholesky decomposition of some matrix $A$ (or $AA'$). Each **Factor** fixes:

- A specific fill-reducing permutation
- A choice of which Cholesky algorithm to use (see **analyze()**)
- Whether we are currently working with real numbers or complex

Given a **Factor** object, you can:

- Compute new Cholesky decompositions of matrices that have the same pattern of non-zeros
- Perform 'updates' or 'downdates'

- Access the various Cholesky factors
- Solve equations involving those factors

## Factoring new matrices

`Factor.`**`cholesky_inplace`**(*A, beta=0*)

Updates this Factor so that it represents the Cholesky decomposition of $A + \beta I$, rather than whatever it contained before.

$A$ must have the same pattern of non-zeros as the matrix used to create this factor originally.

`Factor.`**`cholesky_AAt_inplace`**(*A, beta=0*)

The same as **`cholesky_inplace()`**, except it factors $AA' + \beta I$ instead of $A + \beta I$.

`Factor.`**`cholesky`**(*A, beta=0*)

The same as **`cholesky_inplace()`** except that it first creates a copy of the current **`Factor`** and modifes the copy.

**Returns:** The new **`Factor`** object.

`Factor.`**`cholesky_AAt`**(*A, beta=0*)

The same as **`cholesky_AAt_inplace()`** except that it first creates a copy of the current **`Factor`** and modifes the copy.

**Returns:** The new **`Factor`** object.

## Updating/Downdating

`Factor.`**`update_inplace`**(*C, subtract=False*)

Incremental building of $AA'$ decompositions.

Updates this factor so that instead of representing the decomposition of $A$ ($AA'$), it computes the decomposition of $A + CC'$ ($AA' + CC'$) for `subtract=False` which is the default, or $A - CC'$ ($AA' - CC'$) for `subtract=True`. This method does not require that the **`Factor`** was created with **`cholesky_AAt()`**, though that is the common case.

The usual use for this is to factor AA' when A has a large number of columns, or those columns become available incrementally. Instead of loading all of A into memory, one can load in 'strips' of columns and pass them to this method one at a time.

Note that no fill-reduction analysis is done; whatever permutation was chosen by the initial call to **`analyze()`** will be used regardless of the pattern of non-zeros in C.

## Accessing Cholesky factors explicitly

### Note:

When possible, it is generally more efficient to use the `solve_...` functions documented below rather than extracting the Cholesky factors explicitly.

`Factor.`**`P`**()

Returns the fill-reducing permutation P, as a vector of indices.

The decomposition $LL'$ or $LDL'$ is of:

```
A[P[:, np.newaxis], P[np.newaxis, :]]
```

(or similar for AA').

`Factor.D`()

Converts this factorization to the style

$$LDL' = PAP'$$

or

$$LDL' = PAA'P'$$

and then returns the diagonal matrix D *as a 1d vector*.

> **Note:**
> This method uses an efficient implementation that extracts the diagonal D directly from CHOLMOD's internal representation. It never makes a copy of the factor matrices, or actually converts a full *LL'* factorization into an *LDL'* factorization just to extract *D*.

`Factor.L`()

If necessary, converts this factorization to the style

$$LL' = PAP'$$

or

$$LL' = PAA'P'$$

and then returns the sparse lower-triangular matrix L.

> **Warning:**
> The L matrix returned by this method and the one returned by **`L_D()`** are different!

`Factor.LD`()

If necessary, converts this factorization to the style

$$LDL' = PAP'$$

or

$$LDL' = PAA'P'$$

and then returns a sparse lower-triangular matrix "LD", which contains the D matrix on its diagonal, plus the below-diagonal part of L (the actual diagonal of L is all-ones).

See **`L_D()`** for a more convenient interface.

`Factor.L_D`()

If necessary, converts this factorization to the style

$$LDL' = PAP'$$

or

$$LDL' = PAA'P'$$

and then returns the pair (L, D) where L is a sparse lower-triangular matrix and D is a sparse diagonal matrix.

> **Warning:**
> The L matrix returned by this method and the one returned by **`L()`** are different!



## Solving equations

All methods in this section accept both sparse and dense matrices (or vectors) `b`, and return either a sparse or dense `x` accordingly.

All methods in this section act on $LDL'$ factorizations by default. Thus $L$ refers by default to the matrix returned by **`L_D()`**, not that returned by **`L()`** (though conversion is not performed unless necessary).

`Factor.`**`solve_A`**$(b)$
Solves a linear system.

**Parameters:** **b** – right-hand-side
**Returns:** math:*x*, where $Ax = b$ (or $AA'x = b$, if you used **`cholesky_AAt()`**).

**`__call__()`** is an alias for this function, i.e., you can simply call the **`Factor`** object like a function to solve $Ax = b$.

`Factor.`**`__call__`**$(b)$
Alias for **`solve_A()`**.

`Factor.`**`solve_LDLt`**$(b)$
Solves a linear system.

**Parameters:** **b** – right-hand-side
**Returns:** math:*x*, where $LDL'x = b$.

(This is different from **`solve_A()`** because it does not correct for the fill-reducing permutation.)

`Factor.`**`solve_LD`**$(b)$
Solves a linear system.

**Parameters:** **b** – right-hand-side
**Returns:** math:*x*, where $LDx = b$.

`Factor.`**`solve_DLt`**$(b)$
Solves a linear system.

**Parameters:** **b** – right-hand-side
**Returns:** math:*x*, where $DL'x = b$.

`Factor.`**`solve_L`**$(b)$
Solves a linear system.

**Parameters:**
- **b** – right-hand-side
- **use_LDLt_decomposition** – If True, use the $L$ of the $LDL'$ decomposition. If False, use the $L$ of the $LL'$ decomposition.

**Returns:** math:*x*, where $Lx = b$.

`Factor.`**`solve_Lt`**$(b)$
Solves a linear system.

**Parameters:**
- **b** – right-hand-side
- **use_LDLt_decomposition** – If True, use the $L$ of the $LDL'$ decomposition. If False, use the $L$ of the $LL'$ decomposition.

**Returns:** math:*x*, where $L'x = b$.

`Factor.`**`solve_D`**$(b)$
Returns $x$, where $Dx = b$.

v: latest

`Factor.`**`apply_P`**$(b)$

Returns $x$, where $x = Pb$.

`Factor.`**`apply_Pt`**$(b)$
Returns $x$, where $x = P'b$.

# Convenience methods

`Factor.`**`logdet`**()
Computes the (natural) log of the determinant of the matrix A.

If *f* is a factor, then *f.logdet()* is equivalent to *np.sum(np.log(f.D()))*.

*New in version 0.2.*

`Factor.`**`det`**()
Computes the determinant of the matrix A.

Consider using **`logdet()`** instead, for improved numerical stability. (In particular, determinants are often prone to problems with underflow or overflow.)

*New in version 0.2.*

`Factor.`**`slogdet`**()
Computes the log-determinant of the matrix A, with the same API as **`numpy.linalg.slogdet()`**.

This returns a tuple *(sign, logdet)*, where *sign* is always the number 1.0 (because the determinant of a positive-definite matrix is always a positive real number), and *logdet* is the (natural) logarithm of the determinant of the matrix A.

*New in version 0.2.*

`Factor.`**`inv`**()
Returns the inverse of the matrix A, as a sparse (CSC) matrix.

> **Warning:**
> For most purposes, it is better to use **`solve()`** instead of computing the inverse explicitly. That is, the following two pieces of code produce identical results:
>
> ```
> x = f.solve(b)
> x = f.inv() * b  # DON'T DO THIS!
> ```
>
> But the first line is both faster and produces more accurate results.

Sometimes, though, you really do need the inverse explicitly (e.g., for calculating standard errors in least squares regression), so if that's your situation, here you go.

*New in version 0.2.*

`Factor.`**`copy`**()
Copies the current **`Factor`**.

**Returns:** A new **`Factor`** object.

# Error handling

*class* `sksparse.cholmod.`**`CholmodError`**

*class* `sksparse.cholmod.`**`CholmodNotPositiveDefiniteError`**

*class* `sksparse.cholmod.`**`CholmodNotInstalledError`**

*class* sksparse.cholmod.**CholmodOutOfMemoryError**

*class* sksparse.cholmod.**CholmodTooLargeError**

*class* sksparse.cholmod.**CholmodNotPositiveDefiniteError**

*class* sksparse.cholmod.**CholmodInvalidError**

*class* sksparse.cholmod.**CholmodGpuProblemError**

Errors detected by CHOLMOD or by our wrapper code are converted into exceptions of type **CholmodError** or an appropriated subclass.

*class* sksparse.cholmod.**CholmodWarning**

Warnings issued by CHOLMOD are converted into Python warnings of type **CholmodWarning**.

*class* sksparse.cholmod.**CholmodTypeConversionWarning**

CHOLMOD itself supports matrices in CSC form with 32-bit integer indices and 'double' precision floats (64-bits, or 128-bits total for complex numbers). If you pass some other sort of matrix, then the wrapper code will convert it for you before passing it to CHOLMOD, and issue a warning of type **CholmodTypeConversionWarning** to let you know that your efficiency is not as high as it might be.

> Warning:
> Not all conversions currently produce warnings. This is a bug.

Child of **CholmodWarning**.