scikit-sparse Documentation

Release 0.5.0

Antony Lee

Feb 25, 2026

CONTENTS

Scikit-Sparse is a collection of sparse matrix extensions for SciPy, with a focus on factorization routines and reordering methods.

The package is a thin wrapper around the SuiteSparse library, to make it compatible with `scipy.sparse` arrays.

CHAPTER

ONE

FEATURES

- Fill-reducing orderings: AMD and COLAMD
- Cholesky factorization via CHOLMOD
- Integration with SciPy sparse arrays

CHAPTER

TWO

INSTALLATION

```
conda install -c conda-forge scikit-sparse
# or
pip install scikit-sparse
```

CHAPTER

THREE

QUICK EXAMPLE

```python
import numpy as np
from scipy.sparse import csc_array
from sksparse.cholmod import cho_factor

# Create a sparse positive definite matrix
A = csc_array([[4, 1, 0],
               [1, 3, 0],
               [0, 0, 2]])

# Perform Cholesky factorization
f = cho_factor(A)

# Solve Ax = b
b = np.array([1, 2, 3])
x = f.solve(b)
```

CHAPTER

FOUR

LEARN MORE

- *Overview* - Introduction and installation instructions
- *User Guide* - Tutorials and examples
- *API Reference* - Detailed API documentation
- *Change Log* - List of changes by version
- *Contributing* - Guidelines for contributing to the project

4.1 Overview

4.1.1 Introduction

The `scikit-sparse` package (previously known as `scikits.sparse`) is a companion to the `scipy.sparse` library for sparse matrix manipulation in Python. All `sksparse` routines expect and return `scipy.sparse` matrices (usually in CSC format). The intent of `sksparse` is to wrap code with a GPL license, such as SuiteSparse, which cannot be included in SciPy proper.

4.1.2 Requirements

Installing `scikit-sparse` requires:

- Python >= 3.10
- NumPy >= 2.0
- SciPy >= 1.14
- Cython >= 3.0
- SuiteSparse >= 7.4.0

Older versions may work but are untested.

4.1.3 Installation

Installing SuiteSparse

To install `scikit-sparse`, you need to have the SuiteSparse library installed on your system.

It is recommended that you install SuiteSparse and the scikit-sparse dependencies in a virtual environment, to avoid conflicts with other packages. We recommend using Anaconda:

```
$ conda create -n scikit-sparse python>=3.10 suitesparse
$ conda activate scikit-sparse
```

If you are not using Anaconda, you can install SuiteSparse using your preferred package manager.

On MacOS, you can use Homebrew:

```
$ brew install suite-sparse
```

On Debian/Ubuntu systems, use the following command:

```
$ sudo apt-get install python-scipy libsuitesparse-dev
```

On Arch Linux, run:

```
$ sudo pacman -S suitesparse
```

Installing Scikit-Sparse

Once you have SuiteSparse installed, you can install `scikit-sparse` with:

```
$ conda install -c conda-forge scikit-sparse
```

or if you prefer to use pip, you can install it with:

```
$ pip install scikit-sparse
```

Check if the installation was successful by running the following command:

```
$ python -c "import sksparse; print(sksparse.__version__)"
```

Troubleshooting

The installation will automatically detect the SuiteSparse library and compile the necessary Cython code. It will check for the SuiteSparse library in the following order:

1. The environment variables `SUITESPARSE_INCLUDE_DIR` and `SUITESPARSE_LIB_DIR` (if set, these will override the default search paths)
2. Your active conda environment path
3. Your homebrew paths (*e.g.* `/opt/homebrew/include/suitesparse`)
4. Typical system paths (*e.g.* `/usr/include/suitesparse` on Linux, or `/usr/local/include/suitesparse` on macOS)

The first path that contains the SuiteSparse headers and libraries will be used.

To see which SuiteSparse library was found, you can run the following command on MacOS or Linux:

```
$ CHECK_SKSPARSE_INSTALL=$(python -c 'import sksparse.cholmod; print(sksparse.cholmod.__
↪file__)')
```

Then, use one of the following commands depending on your operating system.

MacOS

On MacOS, use the following command to check where the SuiteSparse installation was found:

```
$ otool -L $CHECK_SKSPARSE_INSTALL | grep cholmod
```

Look for a line that contains `cholmod.*\.dylib` or `cholmod.*\.a`. The output might be something like:

```
$ otool -L $CHECK_SKSPARSE_INSTALL | grep cholmod
/Users/username/src/scikit-sparse/sksparse/cholmod.cpython-313-darwin.so:
        @rpath/libcholmod.5.dylib (compatibility version 5.0.0, current version 5.3.1)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1351.0.
↪0)
```

The `@rpath/libcholmod.5.dylib` indicates that the library was found on the relative path. To resolve this path, run:

```
$ otool -l @rpath/libcholmod.5.dylib | grep -A2 LC_RPATH
      cmd LC_RPATH
  cmdsize 72
     path /Users/username/anaconda3/envs/scikit-sparse/lib (offset 12)
```

which indicates that the library was found on the conda path.

Linux

On Linux, use the following commands instead:

```
$ ldd $CHECK_SKSPARSE_INSTALL | grep cholmod
$ readelf -d $CHECK_SKSPARSE_INSTALL | grep -E '(RPATH|RUNPATH)'
0x000000000000001d (RUNPATH)            Library runpath: [/home/user/anaconda3/envs/
↪scikit-sparse/lib]
```

also confirming installation on the conda path.

4.1.4 Contact

Post your suggestions and questions directly to our GitHub Issues page.

4.1.5 Developers

- 2008 David Cournapeau
- 2009–2015 Nathaniel Smith
- 2010 Dag Sverre Seljebotn
- 2014 Leon Barrett
- 2015 Yuri
- 2016–2017 Antony Lee
- 2016 Alex Grigorievskiy
- 2016–2018 Joscha Reimer
- 2021- Justin Ellis
- 2022- Aaron Johnson
- 2025– Bernard Roesler

4.2 Scikit-Sparse User Guide

Scikit-sparse is a collection of sparse matrix algorithms and convenience functions built to work with SciPySparse arrays. It is largely an interface to the parts of the SuiteSparse library by Timothy A. Davis that have a GPL license and are not suitable for inclusion in SciPy proper.

4.2.1 Subpackages and User Guides

Scikit-sparse is organized into submodules corresponding to the submodules of SuiteSparse. These are summarized in the following table:

Subpackage	Description and User Guide
amd	*Approximate Minimum Degree (AMD) Ordering (sksparse.amd)*
btf	*Block Triangular Form (BTF) (sksparse.btf)*
camd	*Constrained Approximate Minimum Degree (CAMD) Ordering (sksparse.camd)*
ccolamd	*Constrained Column Approximate Minimum Degree (CCOLAMD) Ordering (sksparse.ccolamd)*
cholmod	*Cholesky Decomposition (sksparse.cholmod)*
colamd	*Column Approximate Minimum Degree (COLAMD) Ordering (sksparse.colamd)*
klu	*Clark Kent LU Decomposition (sksparse.klu)*
spqr	*Sparse QR Decomposition (sksparse.spqr)*
umfpack	*Unsymmetric Multifrontal LU Decomposition (sksparse.umfpack)*

Approximate Minimum Degree (AMD) Ordering (*sksparse.amd*)

The *sksparse.amd* module provides an interface to the Approximate Minimum Degree (AMD) ordering algorithm for sparse, square matrices.

It exposes the main function of the AMD package, which computes a symmetric ordering of a sparse matrix that minimizes the fill-in of the Cholesky decomposition. The AMD function accepts both real and complex matrices, in any format supported by `scipy.sparse` (CSC format is most efficient).

Quickstart

If A is a sparse, square matrix, then the following code computes the AMD ordering of A:

```
from sksparse.amd import amd
A = ...  # some sparse matrix
p = amd(A)
PAPT = A[p][:, p]
```

to give the permuted matrix PAP^T, where P is the permutation matrix corresponding to the ordering p. If A is not symmetric, then this is the same as AMD computes the ordering of the symbolically symmetric matrix $A + A^T$.

We can then compute the Cholesky decompositions of the original and permuted matrix using *cholesky()*, and compare the number of non-zeros in each:

```
from sksparse.cholmod import cholesky
A_factor = cholesky(A)
PAPT_factor = cholesky(PAPT)
L = A_factor.L()
Lp = PAPT_factor.L()
print("Number of non-zeros in L: ", L.nnz)
print("Number of non-zeros in Lp:", Lp.nnz)
```

The number of non-zeros in the Cholesky factorization of the permuted matrix should be less than or equal to the number of non-zeros in the Cholesky factorization of the original matrix, but this is not guaranteed.

Example

To see the effects of AMD ordering, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its AMD ordering.

```
#!/usr/bin/env python3
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: amd_example.py
#  Created: 2025-07-29 12:33
# ============================================================================

"""An example of using the AMD (Approximate Minimum Degree) algorithm to find
a fill-reducing ordering of a sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
from scipy.io import mmread

from sksparse.amd import amd
from sksparse.cholmod import cholesky

# Load the bcsstk06 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/bcsstk06>)
filepath = Path("data") / "bcsstk06.mtx"
A = mmread(filepath, spmatrix=False).tocsc()  # read the matrix

p = amd(A)          # compute the AMD ordering
PAPT = A[p][:, p]  # apply the ordering to the matrix

# Compute the Cholesky factorization of each matrix
L = cholesky(A, lower=True)
Lp = cholesky(PAPT, lower=True)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("AMD Example: Fill-Reducing Ordering of bcsstk06")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PAPT, markersize=1)
```

(continues on next page)

(continued from previous page)

```
ax.set_title(r"Permuted Matrix $PAP^T$")

ax = axs[1, 0]
ax.spy(L, markersize=1)
ax.set_title(r"Original Cholesky Factor $L$")
ax.set_xlabel(f"{L.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Lp, markersize=1)
ax.set_title(r"Permuted Cholesky Factor $L_p$")
ax.set_xlabel(f"{Lp.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("amd_example.svg")
```

The figure shows the effect of AMD ordering that reduces the fill-in of the Cholesky factorization of a sparse matrix.

Fig. 1: The number of non-zeros in the Cholesky factorization of the original matrix (left) and the permuted matrix (right) using AMD ordering.

AMDInfo Objects

An *AMDInfo* object is a dataclass returned by the *amd()* function when the `return_info` parameter is set to `True`. It contains information about the AMD ordering, including the return status, the number of non-zeros in the Cholesky factorization, and others.

We can use *AMDInfo* objects to compare the number of non-zeros in the Cholesky factorization of the original matrix, without computing it directly:

```
from sksparse.amd import amd
from sksparse.cholmod import cholesky
A = ...  # some sparse matrix
N = A.shape[0]
p, info = amd(A, return_info=True)
PAPT_factor = cholesky(A[p][:, p])
print("nnz in L of A:   ", info.Lnz + N)
print("nnz in L of PAPT:", PAPT_factor.L().nnz)
```

Typically, the *AMDInfo* object is unnecessary, and you can just use the permutation vector returned by *amd()*.

Convenience Methods

The AMD package also provides a convenience function, *amd_default_control()* to get the default control parameters from the AMD package. Most users will not need to use this function, as the default control parameters are used automatically by *amd()*.

Error Handling

Errors raised by the AMD package are converted into Python exceptions. See the *Exceptions and Warnings* for details.

References

- Amestoy, P. R., Davis, T. A., & Duff, I. S. (1996). *An approximate minimum degree ordering algorithm*. SIAM Journal on Matrix Analysis and Applications, 17(4), 886-905. <https://epubs.siam.org/doi/abs/10.1137/S0895479894278952>.

Block Triangular Form (BTF) (*sksparse.btf*)

The *sksparse.btf* module provides efficient implementations of the Block Triangular Form (BTF) ordering algorithm for sparse, square matrices.

It exposes the main functions of the BTF package, which permutes a sparse matrix into upper block triangular form with a zero-free diagonal, or with a maximum number of nonzeros along the diagonal if a zero-free permutation does not exist. The BTF function accepts both real and complex matrices, in any format supported by `scipy.sparse` (CSC format is most efficient).

Quickstart

If A is a sparse, square matrix, then the following code computes the AMD ordering of A:

```
from sksparse.btf import btf
A = ...  # some sparse matrix
p, q, r = btf(A)
PAQ = A[p][:, q]
```

to give the permuted matrix PAQ, where P is the permutation matrix corresponding to the ordering p, and similarly for q. If A is structurally singular, then the permutation vector q will have negative entries denoting the unmatched indices. To get the actual permutation, use

```
import numpy as np
from sksparse.btf import btf_q_permutation
q_idx = np.nonzero(q < 0)[0]  # store the indices of negative entries
q = btf_q_permutation(q)      # get the permutation vector
PAQ = A[p][:, q]              # permute the matrix
```

Example

To see the effects of BTF ordering, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
#!/usr/bin/env python3
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: btf_example.py
#  Created: 2025-08-07 14:03
# ============================================================================
```

(continues on next page)

(continued from previous page)

```
"""An example of using the BTF (Block Triangular Form) algorithm."""

from pathlib import Path

import matplotlib.pyplot as plt
from scipy.io import mmread

from sksparse.btf import btf, btf_q_permutation, maxtrans, strongcomp


def plot_btf(A, p, q, r, ax=None):
    """Plot the blocks of the BTF matrix.

    Adapted from the SuiteSparse ``drawbtf.m`` [#drawbtf]_.

    Parameters
    ----------
    A : (N, N) sparse matrix
        A square, sparse matrix in CSC format.
    p, q : (N,) ndarray of int
        The row and column permutations that put ``A`` into upper block
        triangular form.
    r : (Nb + 1,) ndarray of int
        The indices of the block boundaries in the permuted matrix. The number
        of blocks is ``len(r) - 1``.
    ax : matplotlib.axes.Axes, optional
        The axes to plot on. If None, the current axes will be used.

    Returns
    -------
    ax : matplotlib.axes.Axes
        The axes with the BTF blocks plotted.

    References
    ----------
    .. [#drawbtf] MATLAB BTF plotting function:
        https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/BTF/MATLAB/drawbtf.m
    """
    if ax is None:
        ax = plt.gca()

    Nb = len(r) - 1
    q = btf_q_permutation(q)  # ensure q is a valid permutation

    for k in range(Nb):
        r0 = r[k]
        r1 = r[k + 1]
        Nk = r1 - r0
        if Nk > 1:
            # Plot the block
            ax.add_patch(
```

(continues on next page)

(continued from previous page)

```
                plt.Rectangle(
                    (r0 - 0.5, r0 - 0.5),
                    Nk,
                    Nk,
                    fill=False,
                    edgecolor="C3",
                    linewidth=1.5,
                )
            )

    return ax


# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False).tocsc()  # read the matrix

# Compute the BTF ordering
p, q, r = btf(A)
q = btf_q_permutation(q)
PAQ = A[p][:, q]

# Compute the maximum transversal
qm = maxtrans(A)
A_mt = A[:, qm]

# Compute the strongly connected components
pcc, rcc = strongcomp(A)
A_scc = A[pcc][:, pcc]

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("BTF Example: Block Triangular Ordering of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PAQ, markersize=1)
plot_btf(PAQ, p, q, r, ax=ax)
ax.set_title(r"BTF Matrix $PAQ$")

ax = axs[1, 0]
ax.spy(A_mt, markersize=1)
ax.set_title(r"Maximum Transveral")
```

(continues on next page)

(continued from previous page)

```
ax = axs[1, 1]
ax.spy(A_scc, markersize=1)
plot_btf(A_scc, pcc, pcc, rcc, ax=ax)
ax.set_title(r"Strongly Connected Components")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("btf_example.svg")
```

This figure shows the effect of BTF ordering:

Convenience Methods

The BTF package also provides a convenience function, *btf_q_permutation()*, to get the actual `q` permutation vector.

Constrained Approximate Minimum Degree (CAMD) Ordering (*sksparse.camd*)

The *sksparse.camd* module provides an interface to the Approximate Minimum Degree (AMD) ordering algorithm for sparse, square matrices.

It exposes the main function of the CAMD package, which computes a symmetric ordering of a sparse matrix that minimizes the fill-in of the Cholesky decomposition. The CAMD function accepts both real and complex matrices, in any format supported by `scipy.sparse` (CSC format is most efficient).

This function is identical to that of the AMD package (*sksparse.amd*), except that it allows the user to specify a set of constraints on the ordering of the matrix.

Quickstart

If A is a sparse, square matrix, then the following code computes the CAMD ordering of A:

```
from sksparse.camd import camd
A = ...  # some sparse matrix
# Set some constraints, e.g., to fix the first K rows and columns
N = A.shape[0]
K = N // 2                    # number of constrained variables
C = np.full(N, K)             # none are constrained (all == K)
C[:K] = np.arange(K)          # first K variables are constrained
p = camd(A, constraints=C)
PAPT = A[p][:, p]
```

to give the permuted matrix PAP^T, where P is the permutation matrix corresponding to the ordering p. If A is not symmetric, then this is the same as CAMD computes the ordering of the symbolically symmetric matrix $A + A^T$.

We can then continue from above to compute the Cholesky decompositions of the original and permuted matrix using *cholesky()*, and compare the number of non-zeros in each:

```
from sksparse.cholmod import cholesky
A_factor = cholesky(A)
PAPT_factor = cholesky(PAPT)
L = A_factor.L()
Lp = PAPT_factor.L()
print("Number of non-zeros in L: ", L.nnz)
print("Number of non-zeros in Lp:", Lp.nnz)
```

The number of non-zeros in the Cholesky factorization of the permuted matrix should be less than or equal to the number of non-zeros in the Cholesky factorization of the original matrix, but this is not guaranteed.

Example

To see the effects of CAMD ordering, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its CAMD ordering.

```
#!/usr/bin/env python3
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ==============================================================================
#     File: camd_example.py
#  Created: 2025-09-30 10:16
# ==============================================================================

"""An example of using the AMD (Approximate Minimum Degree) algorithm to find
a fill-reducing ordering of a sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from scipy.io import mmread

from sksparse.camd import camd
from sksparse.cholmod import cholesky

# Load the bcsstk06 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/bcsstk06>)
filepath = Path("data") / "bcsstk06.mtx"
A = mmread(filepath, spmatrix=False).tocsc()  # read the matrix

N = A.shape[0]
K = N // 2                    # number of constrained variables
C = np.full(N, K)             # fully constrained
C[:K] = np.arange(K)          # first K variables are constrained
Nc = np.count_nonzero(C < K)  # number of constrained variables

p = camd(A, constraints=C)  # compute the AMD ordering
PAPT = A[p][:, p]           # apply the ordering to the matrix
```

(continues on next page)

(continued from previous page)

```
# Compute the Cholesky factorization of each matrix
L = cholesky(A, lower=True)
Lp = cholesky(PAPT, lower=True)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle(
    f"CAMD Example: Fill-Reducing Ordering of bcsstk06\n{Nc} constrained variables"
)

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PAPT, markersize=1)
ax.set_title(r"Permuted Matrix $PAP^T$")

ax = axs[1, 0]
ax.spy(L, markersize=1)
ax.set_title(r"Original Cholesky Factor $L$")
ax.set_xlabel(f"{L.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Lp, markersize=1)
ax.set_title(r"Permuted Cholesky Factor $L_p$")
ax.set_xlabel(f"{Lp.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("camd_example.svg")
```

The figure shows the effect of CAMD ordering that reduces the fill-in of the Cholesky factorization of a sparse matrix, while constraining some of the rows.

Fig. 2: The number of non-zeros in the Cholesky factorization of the original matrix (left) and the permuted matrix (right) using CAMD ordering.

CAMDInfo Objects

An *CAMDInfo* object is a dataclass returned by the *camd()* function when the `return_info` parameter is set to `True`. It contains information about the CAMD ordering, including the return status, the number of non-zeros in the Cholesky factorization, and others.

We can use *CAMDInfo* objects to compare the number of non-zeros in the Cholesky factorization of the original matrix, without computing it directly:

```
from sksparse.camd import camd
from sksparse.cholmod import cholesky
A = ...  # some sparse matrix
N = A.shape[0]
p, info = camd(A, return_info=True)
PAPT_factor = cholesky(A[p][:, p])
print("nnz in L of A:   ", info.Lnz + N)
print("nnz in L of PAPT:", PAPT_factor.L().nnz)
```

Typically, the *CAMDInfo* object is unnecessary, and you can just use the permutation vector returned by *camd()*.

Convenience Methods

The CAMD package also provides a convenience function, *camd_default_control()* to get the default control parameters from the CAMD package. Most users will not need to use this function, as the default control parameters are used automatically by *camd()*.

Error Handling

Errors raised by the CAMD package are converted into Python exceptions. See the *Exceptions and Warnings* for details.

References

- Amestoy, P. R., Davis, T. A., & Duff, I. S. (1996). *An approximate minimum degree ordering algorithm*. SIAM Journal on Matrix Analysis and Applications, 17(4), 886-905. <https://epubs.siam.org/doi/abs/10.1137/S0895479894278952>.

Constrained Column Approximate Minimum Degree (CCOLAMD) Ordering (*sksparse.ccolamd*)

The *sksparse.ccolamd* module provides efficient an implementation of the Column Approximate Minimum Degree (CCOLAMD) ordering algorithm for sparse matrices.

It exposes the main functions of the CCOLAMD package, which computes a column ordering Q of a sparse matrix that minimizes the fill-in of the Cholesky decomposition of $(AQ)^\top(AQ)$. The *ccolamd()* function is appropriate for use with non-symmetric and non-square matrices, for LU factorization, QR factorization, and other decompositions that require a column ordering.

This module also provides a symmetric variant, *csymamd()*, which computes a permutation P of a symmetric matrix A such that the Cholesky factorization of $PAP^\top$ has less fill-in and requires fewer floating point operations than A. This function assumes that its input is symmetric.

The *ccolamd()* and *csymamd()* functions accept both real and complex matrices, in any format supported by `scipy.sparse` (CSC format is most efficient).

These function are identical to that of the COLAMD package (*sksparse.colamd*), except that they allow the user to specify a set of constraints on the ordering of the matrix.

Quickstart

If A is a sparse matrix, then the following code computes the CCOLAMD ordering of A:

```
from sksparse.ccolamd import ccolamd
A = ...  # some sparse matrix
# Set some constraints, e.g., to fix the first K rows and columns
N = A.shape[0]
K = N // 2                    # number of constrained variables
C = np.full(N, K)             # none are constrained (all == K)
C[:K] = np.arange(K)          # first K variables are constrained
q = ccolamd(A, constraints=C)
AQ = A[:, q]                  # permute the columns of A
```

to give the permuted matrix AQ, where Q is the permutation matrix corresponding to the ordering q.

We can then continue from above to compute the LU decompositions of the original and permuted matrix, and compare the number of non-zeros in each:

```
from scipy.sparse.linalg import splu
lu = splu(A)
luq = splu(A[:, q])
L, U = lu.L, lu.U
Lq, Uq = luq.L, luq.U
print("Number of non-zeros in L + U:  ", (L + U).nnz)
print("Number of non-zeros in Lq + Uq:", (Lq + Uq).nnz)
```

The number of non-zeros in the LU factorization of the permuted matrix should be less than or equal to the number of non-zeros in the LU factorization of the original matrix, but this is not guaranteed.

CCOLAMDStats Objects

An *CCOLAMDStats* object is a dataclass returned by the *ccolamd()* function when the `return_info` parameter is set to `True`. It contains information about the ordering, including the return status.

Typically, the *CCOLAMDStats* object is unnecessary, and you can just use the permutation vector returned by *ccolamd()*.

Convenience Methods

The CCOLAMD package also provides a convenience function, *ccolamd_get_defaults()* to get the default control parameters from the CCOLAMD package. Most users will not need to use this function, as the default control parameters are used automatically by *ccolamd()*.

Error Handling

Errors raised by the CCOLAMD package are converted into Python exceptions. See the *Exceptions and Warnings* for details.

Cholesky Decomposition (*sksparse.cholmod*)

The *sksparse.cholmod* module provides an interface to the SuiteSparse CHOLMOD package, which computes basic linear algebra operations for sparse, symmetric, positive-definite matrices.

The main function of this module is to compute the Cholesky factor L of a sparse, symmetric (Hermitian if complex),

positive-definite matrix A with a fill-reducing permutation P, such that:

$$LL^\top = PAP^\top.$$

For matrices that are symmetric but may be numerically close to semi-definite, the module can compute the LDL factorization:

$$LDL^\top = PAP^\top.$$

Either of these factors can then be used to solve linear systems of the form $Ax = b$.

The *cholmod* module exposes most of the capabilities of the CHOLMOD package including:

- Computation of the Cholesky factor with fill-reducing permutation for both real and complex sparse matrices A, in any format supported by `scipy.sparse`.
- An interface for using this decomposition to solve problems of the form $Ax = b$.
- Functions to compute a rank-k "update" or "downdate" of the Cholesky factor.
- The ability to perform the fill-reduction analysis once, and then re-use it to efficiently decompose many matrices with the same pattern of non-zero entries.

This wrapper handles both 32-bit and 64-bit integer types, depending on the input matrix format.

Quickstart

If A is a sparse, symmetric, positive-definite matrix, then the following code computes the Cholesky decomposition of A:

```
from sksparse.cholmod import cholesky
L = cholesky(A)
```

or, with a fill-reducing permutation:

```
L, p = cholesky(A, order="default")
```

See the *example* below for a demonstration of the effect of the fill-reducing permutation.

Once the factorization has been computed, it can be used to solve a linear system:

```
from sksparse.cholmod import ldl_factor
A = ...             # a symmetric, positive-definite sparse matrix
b = ...             # right-hand side
f = ldl_factor(A)   # compute LDL^T factorization
x = f.solve(b)      # solve Ax = b
```

Examples

Cholesky Example

To see how to use the Cholesky factorization, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
```

(continues on next page)

(continued from previous page)

```
#
# ============================================================================
#     File: cholmod_example.py
#  Created: 2025-08-12 20:12
# ============================================================================

"""An example of using CHOLMOD to compute the Cholesky factorization of a
sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
from numpy.testing import assert_allclose
from scipy.io import mmread

from sksparse.cholmod import cholesky

# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False)  # read the matrix
A = (A.T @ A).tocsc()  # make it symmetric positive definite

# compute the Cholesky factorization
R = cholesky(A)
Rp, p = cholesky(A, order="amd")

PAPT = A[p][:, p]  # apply the ordering to the matrix

# Make sure the factorization is correct
assert_allclose((R.T.conj() @ R).toarray(), A.toarray(), atol=1e-9)
assert_allclose((Rp.T.conj() @ Rp).toarray(), PAPT.toarray(), atol=1e-9)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("CHOLMOD Example: Cholesky Factor of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PAPT, markersize=1)
ax.set_title(r"Permuted Matrix $PAP^{\top}$")

ax = axs[1, 0]
ax.spy(R, markersize=1)
```

(continues on next page)

(continued from previous page)

```
ax.set_title(r"Original Cholesky Factor $R$")
ax.set_xlabel(f"{R.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Rp, markersize=1)
ax.set_title(r"Permuted R Factor $R_p$")
ax.set_xlabel(f"{Rp.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("cholesky_example.svg")
```

This figure shows the effect of AMD ordering that reduces the fill-in of the Cholesky factorization of a sparse matrix.

Fig. 3: The number of non-zeros in the Cholesky factorization of the original matrix (left) and the permuted matrix (right) using AMD ordering.

Nested Dissection Example

To see the effects of nested dissection ordering, we can load a sparse matrix and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: nesdis_example.py
#  Created: 2025-08-22 11:04
# ============================================================================

"""An example of using CHOLMOD to compute the Cholesky factorization of a
sparse matrix with various orderings.
"""

from pathlib import Path

import matplotlib.pyplot as plt
from scipy.io import mmread
from scipy.sparse.linalg import splu

from sksparse.amd import amd
from sksparse.cholmod import nesdis

# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
```

(continues on next page)

(continued from previous page)

```
A = mmread(filepath, spmatrix=False).tocsc()  # read the matrix

# Compute the AMD permutation
p = amd(A)
PAPT = A[p][:, p]

# Compute the nested dissection A.T @ A permutation
q = nesdis(A)
QAQT = A[q][:, q]

# Compute the LU decompositions
lu = splu(A)
lup = splu(PAPT)
luq = splu(QAQT)

LU = lu.L + lu.U
LUp = lup.L + lup.U
LUq = luq.L + luq.U

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=3, ncols=2, clear=True)
fig.set_size_inches((6, 10), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("CHOLMOD Example: Cholesky Factor of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[1, 0]
ax.spy(PAPT, markersize=1)
ax.set_title(r"AMD-Ordered Matrix $PAP^{\top}$")

ax = axs[0, 1]
ax.spy(LU, markersize=1)
ax.set_title(r"LU Factors $LU = A$")
ax.set_xlabel(f"{LU.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(LUp, markersize=1)
ax.set_title(r"AMD Factors $LU = PAP^{\top}$")
ax.set_xlabel(f"{LUp.nnz:,} non-zeros")

ax = axs[2, 0]
ax.spy(QAQT, markersize=1)
ax.set_title(r"Nested Dissection Matrix $QAQ^{\top}$")

ax = axs[2, 1]
ax.spy(LUq, markersize=1)
ax.set_title(r"Nesdis Factors $LU = QAQ^{\top}$")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(f"{LUq.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("nesdis_example.svg")
```

This figure shows the effect of nested dissection ordering that reduces the fill-in of the LU factorization of a sparse matrix in a case where the AMD order *does not* help.

Fig. 4: The number of non-zeros in the LU factorization of the original matrix and the permuted matrix using AMD and nested dissection ordering.

Function Interface

For users who want to directly compute the factorization without needing to manipulate the *CholeskyFactor* object, the *cholmod* module provides the *cholesky()* and *ldl()* functions that perform both the symbolic analysis and the numerical factorization in one step, and return the matrices directly.

Object Interface

For more advanced usage, users can instantiate the *CholeskyFactor* class. This class can be instantiated directly using its constructor, or more conveniently using the *cho_factor()* or *ldl_factor()* functions.

When instantiated directly, the constructor performs a symbolic analysis of the matrix, but does not compute the numerical factorization. The numerical factorization is then performed by calling the *CholeskyFactor.factorize()* method.

The *cho_factor()* and *ldl_factor()* functions perform both the symbolic analysis and the numerical factorization in one step, and return an instance of the *CholeskyFactor* class.

The resulting *CholeskyFactor* object can then be used to solve linear systems using its *CholeskyFactor.solve()* method, or to update the factorization in-place using the *update()*, *rowadd()*, *rowdel()*, and *resymbol()* methods.

The *CholeskyFactor.factorize()* method can be called again to factor a new matrix with the same sparsity pattern.

Symbolic Analysis

In addition to numerical factorization, *cholmod* provides symbolic operations *symbfact()*, and *etree()* that can be used to analyze the structure of the Cholesky factor and to compute fill-reducing permutations.

Graph Partitioning

The *cholmod* module also includes functions for graph partitioning and node reordering, which can be used like the *amd* and *colamd* (and their constrained counterparts) modules to reduce fill-in during factorization.

These functions provide a direct interface to the corresponding CHOLMOD functions that are used internally by *cholesky()* and *ldl()* when the `order` argument is specified. The functions *bisect()*, *metis()*, and *nesdis()* can be used to compute fill-reducing orderings, and the *SeparatorTree* class represents the resulting separator tree.

Exceptions and Warnings

Warnings issued by CHOLMOD are converted into Python warnings of type *CholmodWarning*. The module will also issue a SparseEfficiencyWarning if the input matrix is not a csc_array (note that the csc_matrix class is not supported, as it will be deprecated and is not recommended for use in new code).

Errors detected by CHOLMOD or by our wrapper code are converted into exceptions of type *CholmodError* or an appropriate subclass. See the *Exceptions and Warnings* for details.

Column Approximate Minimum Degree (COLAMD) Ordering (*sksparse.colamd*)

The *sksparse.colamd* module provides efficient an implementation of the Column Approximate Minimum Degree (COLAMD) ordering algorithm for sparse matrices.

It exposes the main functions of the COLAMD package, which computes a column ordering Q of a sparse matrix that minimizes the fill-in of the Cholesky decomposition of $(AQ)^\top(AQ)$. The *colamd()* function is appropriate for use with non-symmetric and non-square matrices, for LU factorization, QR factorization, and other decompositions that require a column ordering.

This module also provides a symmetric variant, *symamd()*, which computes a permutation P of a symmetric matrix A such that the Cholesky factorization of $PAP^\top$ has less fill-in and requires fewer floating point operations than A. This function assumes that its input is symmetric.

The *colamd()* and *symamd()* functions accept both real and complex matrices, in any format supported by scipy.sparse (CSC format is most efficient).

Quickstart

If A is a sparse matrix, then the following code computes the COLAMD ordering of A:

```
from sksparse.colamd import colamd
A = ...  # some sparse matrix
q = colamd(A)
AQ = A[:, q]  # permute the columns of A
```

to give the permuted matrix AQ, where Q is the permutation matrix corresponding to the ordering q.

We can then continue from above to compute the LU decompositions of the original and permuted matrix, and compare the number of non-zeros in each:

```
from scipy.sparse.linalg import splu
lu = splu(A)
luq = splu(A[:, q])
L, U = lu.L, lu.U
Lq, Uq = luq.L, luq.U
print("Number of non-zeros in L + U:  ", (L + U).nnz)
print("Number of non-zeros in Lq + Uq:", (Lq + Uq).nnz)
```

The number of non-zeros in the LU factorization of the permuted matrix should be less than or equal to the number of non-zeros in the LU factorization of the original matrix, but this is not guaranteed.

Example

To see the effects of COLAMD ordering, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: colamd_example.py
#  Created: 2025-07-31 14:18
# ============================================================================

"""An example of using the COLAMD (Column Approximate Minimum Degree) algorithm
to find a fill-reducing ordering of a sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
from scipy.io import mmread
from scipy.sparse.linalg import splu

from sksparse.colamd import colamd

# from numpy.testing import assert_allclose, assert_array_equal


# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False).tocsc()  # read the matrix

q = colamd(A)  # compute the COLAMD ordering
AQ = A[:, q]   # apply the column ordering to the matrix

# Compute the LU factorization of the original and permuted matrices
lu = splu(A, permc_spec="NATURAL")
L_, U_ = lu.L, lu.U

luq = splu(AQ, permc_spec="NATURAL")
Lq, Uq = luq.L, luq.U

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("COLAMD Example: Fill-Reducing Ordering of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
```

(continues on next page)

(continued from previous page)

```
ax.spy(AQ, markersize=1)
ax.set_title(r"Permuted Matrix $AQ$")

ax = axs[1, 0]
ax.spy(L_ + U_, markersize=1)
ax.set_title(r"Original LU Factors $L + U$")
ax.set_xlabel(f"{(L_ + U_).nnz:,} total non-zeros")

ax = axs[1, 1]
ax.spy(Lq + Uq, markersize=1)
ax.set_title(r"Permuted LU Factors $L_q + U_q$")
ax.set_xlabel(f"{(Lq + Uq).nnz:,} total non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("colamd_example.svg")
```

This figure shows the effect of COLAMD ordering that reduces the fill-in of the Cholesky factorization of a sparse matrix.

Fig. 5: The number of non-zeros in the LU factorization of the original matrix (left) and the permuted matrix (right) using COLAMD ordering.

COLAMDStats Objects

An *COLAMDStats* object is a dataclass returned by the *colamd()* function when the `return_info` parameter is set to `True`. It contains information about the ordering, including the return status.

Typically, the *COLAMDStats* object is unnecessary, and you can just use the permutation vector returned by *colamd()*.

Convenience Methods

The COLAMD package also provides a convenience function, *colamd_get_defaults()* to get the default control parameters from the COLAMD package. Most users will not need to use this function, as the default control parameters are used automatically by *colamd()*.

Error Handling

Errors raised by the COLAMD package are converted into Python exceptions. See the *Exceptions and Warnings* for details.

Clark Kent LU Decomposition (*sksparse.klu*)

The *sksparse.klu* module provides an interface to the SuiteSparse KLU package, which computes basic linear algebra operations for sparse, square matrices.

The main function of this module is to compute the LU factorization of a sparse matrix A with a fill-reducing permutation P and Q, and row-scaling R, and offset F such that:

$$LU + F = RPAQ.$$

These factors can then be used to solve linear systems of the form $Ax = b$.

The *klu* module exposes most of the capabilities of the KLU package including:

- Computation of the LU factors with fill-reducing permutation for both real and complex sparse matrices A, in any format supported by `scipy.sparse`.
- An interface for using this decomposition to solve problems of the form $Ax = b$.
- The ability to perform the fill-reduction analysis once, and then re-use it to efficiently decompose many matrices with the same pattern of non-zero entries.

Quickstart

If A is a sparse matrix then the following code computes the LU decomposition of A:

```
from sksparse.klu import klu_factor
L, U, p, q, r, F, _ = klu_factor(A)
# L @ U + F == r[:, np.newaxis] * A[p[:, np.newaxis], q]
```

See the *example* below for a demonstration of the effect of the fill-reducing permutation.

Once the factorization has been computed, it can be used to solve a square linear system:

```
from sksparse.klu import klu_factor
A = ...  # a sparse, square matrix
b = ...  # right-hand side
x = klu_solve(A, b)
```

Example

To see how to use the LU factorization, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: klu_example.py
#  Created: 2025-11-17 10:38
# ============================================================================

"""An example of using KLU to compute the klu_factor factorization of a
sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from numpy.testing import assert_allclose
from scipy.io import mmread

from sksparse.klu import klu_factor
```

(continues on next page)

(continued from previous page)

```
from sksparse.umfpack import umf_factor

# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False)  # read the matrix
A = A.tocsc()

# Compute the LU factorization
# NOTE that KLU does not support a "natural" ordering, so use UMFPACK here
L, U, p, q, rscale = umf_factor(A, ordering_method="none")
Lp, Up, pp, qp, rscalep, Fp, _ = klu_factor(A, ordering="AMD")

# Make sure the factorization is correct
PRAQ = (rscale[:, np.newaxis] * A).tocsc()[p[:, np.newaxis], q]
PRAQp = rscalep[:, np.newaxis] * A[pp[:, np.newaxis], qp]
assert_allclose((L @ U).toarray(), PRAQ.toarray(), atol=1e-9)
assert_allclose((Lp @ Up + Fp).toarray(), PRAQp.toarray(), atol=1e-9)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("KLU Example: LU Factors of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PRAQ, markersize=1)
ax.set_title(r"Permuted Matrix $PAQ$")

ax = axs[1, 0]
ax.spy(L + U, markersize=1)
ax.set_title(r"Original LU Factors $L + U$")
ax.set_xlabel(f"{L.nnz + U.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Lp + Up, markersize=1)
ax.set_title(r"Permuted LU Factors $L_p + U_p$")
ax.set_xlabel(f"{Lp.nnz + Up.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("klu_example.svg")
```

This figure shows the effect of AMD ordering that reduces the fill-in of the LU factorization of a sparse matrix.

Fig. 6: The number of non-zeros in the LU factorization of the original matrix (left) and the permuted matrix (right) using AMD ordering.

Function Interface

For users who want to directly compute the factorization without needing to manipulate the *KLUFactor* object, the *klu* module provides the *klu_factor()* function that performs both the symbolic analysis and the numeric factorization in one step. It returns the factorization object as a single output, or the individual factors and permutations can be unpacked as multiple outputs.

To directly solve a linear system without explicitly computing the factorization, the *klu_solve()* function can be used. This function performs the factorization internally and returns the solution to the linear system.

Object Interface

For more advanced usage, users can instantiate the *KLUFactor* class. This class can be instantiated directly using its constructor, or more conveniently using the *klu_factor()* function.

When instantiated directly, the constructor performs a symbolic analysis of the matrix, but does not compute the numeric factorization. The numeric factorization is then performed by calling the *KLUFactor.factorize()* method.

The *klu_factor()* function performs both the symbolic analysis and the numeric factorization in one step, and returns an instance of the *KLUFactor* class. The resulting *KLUFactor* object can then be used to solve linear systems using its *KLUFactor.solve()* method. The *KLUFactor.factorize()* method can be called again to factor a new matrix with the same sparsity pattern.

Customization and Statistics

The *klu_factor()* accepts several optional parameters that control the behavior of the factorization, such as the fill-reducing ordering method to use. The *KLUControl* object can be instantiated directly to customize the factorization options before passing it to the *KLUFactor* constructor or the *klu_factor()* function. Any options not explicitly set by the user will be given the KLU default values.

After a factorization is computed, statistics about the factorization can be accessed via the *KLUFactor.info* attribute, which is an instance of the *KLUInfo* class. This object contains various performance metrics and statistics about the factorization process.

Exceptions and Warnings

Warnings issued by KLU are converted into Python warnings of type *KLUWarning*. The module will also issue a `SparseEfficiencyWarning` if the input matrix is not a `csc_array` (note that the `csc_matrix` class is not supported, as it will be deprecated and is not recommended for use in new code).

Errors detected by KLU or by our wrapper code are converted into exceptions of type *KLUError* or an appropriate subclass. See the *Warnings and Exceptions* for details.

Sparse QR Decomposition (*sksparse.spqr*)

The *sksparse.spqr* module provides an interface to the SuiteSparse SPQR package, which computes basic linear algebra operations for sparse matrices.

The main function of this module is to compute the QR factorization of a sparse matrix A with a fill-reducing permutation E such that:

$$QR = AE$$

These factors can then be used to solve linear systems of the form $Ax = b$.

The *spqr* module exposes most of the capabilities of the SPQR package including:

- Computation of the QR factors with fill-reducing permutation for both real and complex sparse matrices A, in any format supported by `scipy.sparse`.
- An interface for using this decomposition to solve problems of the form $Ax = b$.
- The ability to perform the fill-reduction analysis once, and then re-use it to efficiently decompose many matrices with the same pattern of non-zero entries.

Quickstart

If A is a sparse matrix then the following code computes the QR decomposition of A:

```
from sksparse.spqr import spqr_factor
Q, R, p = spqr(A)
# Q @ R == A[:, p]
```

See the *example* below for a demonstration of the effect of the fill-reducing permutation.

Once the factorization has been computed, it can be used to solve a square linear system:

```
from sksparse.spqr import spqr_factor
A = ...  # a sparse matrix
b = ...  # right-hand side
x = spqr_solve(A, b)
```

Example

To see how to use the QR factorization, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
#
# ============================================================================
#     File: spqr_example.py
#  Created: 2025-11-17 11:58
# ============================================================================

"""An example of using SPQR to compute the QR factorization of a
sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
from numpy.testing import assert_allclose
from scipy.io import mmread

from sksparse.spqr import spqr
```

(continues on next page)

(continued from previous page)

```
# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False)  # read the matrix
A = A.tocsc()

# compute the LU factorization
Q, R, _ = spqr(A, order="fixed")
Qp, Rp, p = spqr(A, order="colamd")

# Permute A for visualization
AE = A[:, p]

# Make sure the factorization is correct
assert_allclose((Q @ R).toarray(), A.toarray(), atol=1e-9)
assert_allclose((Qp @ Rp).toarray(), AE.toarray(), atol=1e-9)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=3, ncols=2, clear=True)
fig.set_size_inches((6, 10), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("SPQR Example: QR Factors of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(AE, markersize=1)
ax.set_title(r"Permuted Matrix $AE$")

ax = axs[1, 0]
ax.spy(Q, markersize=1)
ax.set_title(r"Original Q Factor $Q$")
ax.set_xlabel(f"{Q.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Qp, markersize=1)
ax.set_title(r"Permuted Q Factor $Q_p$")
ax.set_xlabel(f"{Qp.nnz:,} non-zeros")

ax = axs[2, 0]
ax.spy(R, markersize=1)
ax.set_title(r"Original R Factor $R$")
ax.set_xlabel(f"{R.nnz:,} non-zeros")

ax = axs[2, 1]
ax.spy(Rp, markersize=1)
ax.set_title(r"Permuted R Factor $R_p$")
ax.set_xlabel(f"{Rp.nnz:,} non-zeros")
```

(continues on next page)

(continued from previous page)

```
for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("spqr_example.svg")
```

This figure shows the effect of COLAMD ordering that reduces the fill-in of the QR factorization of a sparse matrix.

Fig. 7: The number of non-zeros in the QR factorization of the original matrix (left) and the permuted matrix (right) using COLAMD ordering.

Function Interface

For users who just need to compute the QR factorization and return the factors as sparse matrices, the *spqr()* function can be used. This function takes a sparse matrix as input and returns the Q and R factors as sparse matrices, along with the fill-reducing column permutation vector. There is also a `mode` argument that is similar to the one in `scipy.linalg.qr()`, which allows users to specify whether they want the full or reduced factors, just the R factor, or the Householder form of the Q factor.

Typically, the Householder form is substantially less memory-intensive than the explicit Q factor, especially for large sparse matrices. The function *spqr_qmult()* can then be used to efficiently apply the Q factor to a dense matrix or vector without explicitly forming Q.

To directly solve a linear system without explicitly computing the factorization, the *spqr_solve()* function can be used. This function performs the factorization internally and returns the solution to the linear system. It can be used on non-square matrices as well. If A is a matrix of shape M by N, solving an overdetermined system ($M > N$) will return the least-squares solution, while solving an underdetermined system ($M < N$) will return the minimum-norm solution.

Object Interface

For more advanced usage, users can instantiate the *SPQRFactor* class. This class can be instantiated directly using its constructor, or more conveniently using the *spqr_factor()* function.

When instantiated directly, the constructor performs a symbolic analysis of the matrix, but does not compute the numeric factorization. The numeric factorization is then performed by calling the *SPQRFactor.factorize()* method.

The *spqr_factor()* function performs both the symbolic analysis and the numeric factorization in one step, and returns an instance of the *SPQRFactor* class. The resulting *SPQRFactor* object can then be used to solve linear systems using its *SPQRFactor.solve()* method. The *SPQRFactor.factorize()* method can be called again to factor a new matrix with the same sparsity pattern. The *SPQRFactor* class also provides a method to apply the Q factor to a dense matrix or vector, using the *SPQRFactor.qmult()* method.

Currently, it is not possible to extract the Q and R factors as sparse matrices from the *SPQRFactor* object. Users who need the explicit factors should use the *spqr()* function instead.

After a factorization is computed, statistics about the factorization can be accessed via the *SPQRFactor.info* attribute, which is an instance of the *SPQRInfo* class. This object contains various performance metrics and statistics about the factorization process.

Exceptions and Warnings

Warnings issued by SPQR are converted into Python warnings of type *SPQRWarning*. The module will also issue a SparseEfficiencyWarning if the input matrix is not a csc_array (note that the csc_matrix class is not supported, as it will be deprecated and is not recommended for use in new code).

Errors detected by SPQR or by our wrapper code are converted into exceptions of type *SPQRError* or an appropriate subclass. See the *Warnings and Exceptions* for details.

Unsymmetric Multifrontal LU Decomposition (*sksparse.umfpack*)

The *sksparse.umfpack* module provides an interface to the SuiteSparse UMFPACK package, which computes basic linear algebra operations for sparse matrices.

The main function of this module is to compute the LU factorization of a sparse matrix A with a fill-reducing permutation P and Q, and row-scaling R, such that:

$$LU = PRAQ.$$

These factors can then be used to solve linear systems of the form $Ax = b$, or $A^\top x = b$.

The *umfpack* module exposes most of the capabilities of the UMFPACK package including:

- Computation of the LU factors with fill-reducing permutation for both real and complex sparse matrices A, in any format supported by scipy.sparse.
- An interface for using this decomposition to solve problems of the form $Ax = b$.
- The ability to perform the fill-reduction analysis once, and then re-use it to efficiently decompose many matrices with the same pattern of non-zero entries.

Quickstart

If A is a sparse matrix then the following code computes the LU decomposition of A:

```
from sksparse.umfpack import umf_factor
L, U, p, q, r = umf_factor(A)
# L @ U == (r[:, np.newaxis] * A).tocsc()[p[:, np.newaxis], q]
```

See the *example* below for a demonstration of the effect of the fill-reducing permutation.

Once the factorization has been computed, it can be used to solve a square linear system:

```
from sksparse.umfpack import umf_factor
A = ...  # a sparse matrix
b = ...  # right-hand side
x = umf_solve(A, b)
```

Example

To see how to use the LU factorization, we can load a sparse matrix from the SuiteSparse Matrix Collection and compute its ordering.

```
# Part of the scikit-sparse project.
# Copyright (C) 2025 Bernard Roesler. All rights reserved.
# See pyproject.toml for full author list and LICENSE.txt for license details.
# SPDX-License-Identifier: BSD-2-Clause
```

(continues on next page)

(continued from previous page)

```
#
# ============================================================================
#     File: umfpack_example.py
#  Created: 2025-11-17 09:35
# ============================================================================

"""An example of using UMFPACK to compute the umf_factor factorization of a
sparse matrix.
"""

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from numpy.testing import assert_allclose
from scipy.io import mmread

from sksparse.umfpack import umf_factor

# Load the west0479 matrix (downloaded from the SuiteSparse Matrix Collection:
# <https://sparse.tamu.edu/HB/west0479>)
filepath = Path("data") / "west0479.mtx"
A = mmread(filepath, spmatrix=False)  # read the matrix

# compute the LU factorization
L, U, p, q, rscale = umf_factor(A, ordering_method="none")
Lp, Up, pp, qp, rscalep = umf_factor(A, ordering_method="amd")

# Make sure the factorization is correct
PRAQ = (rscale[:, np.newaxis] * A).tocsc()[p[:, np.newaxis], q]
PRAQp = (rscalep[:, np.newaxis] * A).tocsc()[pp[:, np.newaxis], qp]
assert_allclose((L @ U).toarray(), PRAQ.toarray(), atol=1e-9)
assert_allclose((Lp @ Up).toarray(), PRAQp.toarray(), atol=1e-9)

# Plot the original and permuted matrices
plt.rcParams.update({"font.size": 10})

fig, axs = plt.subplots(num=1, nrows=2, ncols=2, clear=True)
fig.set_size_inches((6, 6), forward=True)
fig.set_constrained_layout(True)
fig.suptitle("UMFPACK Example: LU Factors of west0479")

ax = axs[0, 0]
ax.spy(A, markersize=1)
ax.set_title(r"Original Matrix $A$")

ax = axs[0, 1]
ax.spy(PRAQ, markersize=1)
ax.set_title(r"Permuted Matrix $PAQ$")

ax = axs[1, 0]
ax.spy(L + U, markersize=1)
```

(continues on next page)

(continued from previous page)

```
ax.set_title(r"Original LU Factors $L + U$")
ax.set_xlabel(f"{L.nnz + U.nnz:,} non-zeros")

ax = axs[1, 1]
ax.spy(Lp + Up, markersize=1)
ax.set_title(r"Permuted LU Factors $L_p + U_p$")
ax.set_xlabel(f"{Lp.nnz + Up.nnz:,} non-zeros")

for ax in axs.flat:
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
fig.savefig("umfpack_example.svg")
```

This figure shows the effect of AMD ordering that reduces the fill-in of the LU factorization of a sparse matrix.

Fig. 8: The number of non-zeros in the LU factorization of the original matrix (left) and the permuted matrix (right) using AMD ordering.

Function Interface

For users who want to directly compute the factorization without needing to manipulate the *UMFFactor* object, the *umfpack* module provides the *umf_factor()* function that performs both the symbolic analysis and the numeric factorization in one step. It returns the factorization object as a single output, or the individual factors and permutations can be unpacked as multiple outputs.

To directly solve a linear system without explicitly computing the factorization, the *umf_solve()* function can be used. This function performs the factorization internally and returns the solution to the linear system.

Object Interface

For more advanced usage, users can instantiate the *UMFFactor* class. This class can be instantiated directly using its constructor, or more conveniently using the *umf_factor()* function.

When instantiated directly, the constructor performs a symbolic analysis of the matrix, but does not compute the numeric factorization. The numeric factorization is then performed by calling the *UMFFactor.factorize()* method.

The *umf_factor()* function performs both the symbolic analysis and the numeric factorization in one step, and returns an instance of the *UMFFactor* class. The resulting *UMFFactor* object can then be used to solve linear systems using its *UMFFactor.solve()* method. The *UMFFactor.factorize()* method can be called again to factor a new matrix with the same sparsity pattern.

Customization and Statistics

The *umf_factor()* accepts several optional parameters that control the behavior of the factorization, such as the fill-reducing ordering method to use. These options are stored in a *UMFControl* object that can be accessed via the *UMFFactor.control* attribute. The *UMFControl* object can also be instantiated directly to customize the factorization options before passing it to the *UMFFactor* constructor or the *umf_factor()* function. Any options not explicitly set by the user will be given the UMFPACK default values.

After a factorization is computed, statistics about the factorization can be accessed via the *UMFFactor.info* attribute, which is an instance of the *UMFInfo* class. This object contains various performance metrics and statistics about the

factorization process.

Exceptions and Warnings

Warnings issued by UMFPACK are converted into Python warnings of type *UMFPACKWarning*. The module will also issue a `SparseEfficiencyWarning` if the input matrix is not a `csc_array` (note that the `csc_matrix` class is not supported, as it will be deprecated and is not recommended for use in new code).

Errors detected by UMFPACK or by our wrapper code are converted into exceptions of type *UMFPACKError* or an appropriate subclass. See the *Warnings and Exceptions* for details.

4.3 Scikit Sparse API (*sksparse*)

4.3.1 Approximate Minimum Degree (AMD) Ordering (*sksparse.amd*)

Added in version 0.5.0.

Python interface to the Approximate Minimum Degree (AMD) ordering algorithm.

Interface

AMDInfo(status, N, nz, symmetry, nzdiag, ...)	Information statistics returned by the AMD algorithm.
amd(A[, dense_thresh, aggressive, return_info])	Compute the approximate minimum degree ordering of a sparse matrix.
amd_default_control()	Get the default control parameters for AMD.

AMDInfo

class sksparse.amd.**AMDInfo**(*status: int*, *N: int*, *nz: int*, *symmetry: float*, *nzdiag: int*, *nz_A_plus_AT: int*, *Ndense: int*, *memory: float*, *Ncmpa: int*, *Lnz: int*, *Ndiv: int*, *Nmultsubs_LDL: int*, *Nmultsubs_LU: int*, *dmax: int*)

Information statistics returned by the AMD algorithm.

This class wraps the contents of the `Info` array output by `amd_order()` into a Python dataclass.

status

Return status:

- 0 = OK,
- 1 = OK but jumbled,
- -1 = out of memory,
- -2 = invalid matrix.

Type
int

N

Number of rows and columns of the input matrix `A`.

Type
int

nz

Number of nonzeros in the input matrix `A`.

Type
int

symmetry

Symmetry of pattern of `A`. The symmetry is the number of "matched" off-diagonal entries divided by the total number of off-diagonal entries. An entry `A[i, j]` is matched if `A[j, i]` is also an entry, for any pair `[i, j]` where `i != j`. In python code:

```
S = A.astype(bool)
B = sparse.tril(S, -1) + sparse.triu(S, 1)
symmetry = (B * B.T).nnz / B.nnz
```

Type
`float` $\in [0, 1]$

nzdiag

Number of entries on the diagonal of `A`.

Type
int

nz_A_plus_AT

Number of nonzeros in `A + A.T` (excluding diagonal). If `A` is perfectly symmetric (`symmetry = 1`), with a fully non-zero diagonal, then `nz_A_plus_AT = nz - N` (the smallest possible value). If `A` is perfectly unsymmetric (`symmetry = 0`, for an upper triangular matrix, *e.g.*) with no diagonal, then `nz_A_plus_AT = 2 * nz` (the largest possible value).

Type
int

Ndense

Number of dense rows/columns ignored during ordering. These rows/columns are placed last in the output order `p`.

Type
int

memory

Memory used, in bytes. This is equal to: `(1.2 * nz_A_plus_AT + 9 * N) * sizeof(int)`. This coefficient is at most `2.4 * nz + 9 * N`. This accounting excludes the size of the input arguments `Ap`, `Ai`, and `p`, which have a total size of `nz + 2 * N + 1` integers.

Type
float

Ncmpa

Number of components in the matrix (excluding dense rows/columns).

Type
int

Lnz

Number of nonzeros in the Cholesky factor `L` of `A`, excluding the diagonal. This is a slight upper bound because of the approximate degree algorithm. It is a rough upper bound if there are many dense rows/columns. The remaining statistics are also slight or rough upper bounds for the same reason.

Type
int

Ndiv

Number of division operations for LU or Cholesky factorization of the permuted matrix `A[p][:, p]`.

Type
int

Nmultsubs_LDL

Number of multiply-subtract pairs for `LDL.T` factorization.

Type
int

Nmultsubs_LU

Number of multiply-subtract pairs for LU factorization, assuming that no numerical pivoting is required.

Type
int

dmax

Maximum number of nonzeros in any column of `L`, including the diagonal.

Type
int

Notes

Field descriptions are adapted from SuiteSparse `amd.h`[1].

Added in version 0.5.0.

References

Methods

__init__
from_array

__init__

`AMDInfo.__init__`(*status: int*, *N: int*, *nz: int*, *symmetry: float*, *nzdiag: int*, *nz_A_plus_AT: int*, *Ndense: int*, *memory: float*, *Ncmpa: int*, *Lnz: int*, *Ndiv: int*, *Nmultsubs_LDL: int*, *Nmultsubs_LU: int*, *dmax: int*) → None

from_array

classmethod `AMDInfo.from_array`(*info: ndarray*) → *AMDInfo*

amd

[1] `amd.h` - SuiteSparse AMD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/AMD/Include/amd.h

sksparse.amd.**amd**(*A*, *dense_thresh=None*, *aggressive=None*, *return_info=False*)

Compute the approximate minimum degree ordering of a sparse matrix.

Adapted from the SuiteSparse *amd.h* documentation[0]:

> AMD finds a fill-reducing ordering of a sparse matrix `A`, using the approximate minimum degree algorithm. The output is a permutation vector `p` such that the Cholesky factor of `A[p][:, p]` has fewer nonzeros than the Cholesky factor of `A`. If `A` is not symmetric, the algorithm computes an ordering of `A + A.T`.

For more details on the entire package, see the SuiteSparse homepage[1] and Github repository[2].

Parameters

- **A** (*(N, N) array_like or sparse matrix*) – A square matrix in CSC format or convertible to CSC.
- **dense_thresh** (*float, optional*) – Threshold number of entries for considering a row/column dense. If None, use the default value from AMD. The default value is 10.

  Adapted from the SuiteSparse *amd.h* documentation[0]:

  > A dense row/column in `A + A.T` can cause AMD to spend a lot of time in ordering the matrix. If `dense_thresh >= 0`, rows/columns with more than `max(dense_thresh * sqrt(N), 16)` entries are ignored during the ordering, and placed last in the output order. The default value of `dense_thresh` is 10. If negative, no rows/columns are treated as "dense". Rows/columns with 16 or fewer off-diagonal entries are never considered "dense".
- **aggressive** (*bool, optional*) – If True, use aggressive absorption. If None, uses the default value from AMD. The default value is True.

  Adapted from the SuiteSparse *amd.h* documentation[0]:

  > Controls whether or not to use aggressive absorption, in which a prior element is absorbed into the current element if is a subset of the current element, even if it is not adjacent to the current pivot element (refer to Amestoy, Davis, & Duff, 1996, for more details). The default value is `True`, which means to perform aggressive absorption. This nearly always leads to a better ordering (because the approximate degrees are more accurate) and a lower execution time. There are cases where it can lead to a slightly worse ordering, however.
- **return_info** (*bool, optional*) – If True, returns additional information about the ordering process. Default is False.

Returns

- **p** (`ndarray`) – The permutation vector such that the Cholesky factor of `A[p][:, p]` has fewer nonzeros than the Cholesky factor of `A`.
- **info** (`ndarray`, optional) – Additional information about the ordering process, returned if `return_info` is True. Contains various statistics and status codes.

Raises

- `SparseEfficiencyWarning` – If the input matrix is not in CSC format, a warning is raised and the matrix is converted to CSC format.
- `ValueError` – If the input matrix is not square or cannot be converted to CSC format.

[0] *amd.h* - Source header file from SuiteSparse. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/AMD/Include/amd.h
[1] SuiteSparse homepage. https://people.engr.tamu.edu/davis/suitesparse.html
[2] SuiteSparse GitHub repository. https://github.com/DrTimothyAldenDavis/SuiteSparse

- ***AMDInvalidMatrixError*** – If the input matrix is invalid for AMD, such as having unsupported data types or formats.
- ***AMDMemoryError*** – If the AMD algorithm runs out of memory during execution.

See also

camd, *colamd*, *ccolamd*

Notes

This function wraps the AMD (Approximate Minimum Degree) algorithm from the SuiteSparse by Timothy A. Davis. For details, see the SuiteSparse repository[Page 43, 2].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.amd import amd
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(565656)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> p, info = amd(A, return_info=True)
>>> p
array([ 1,  4,  8,  6,  0,  3,  5,  2,  9, 10,  7])
>>> info
AMDInfo(status=0, N=11, nz=43, symmetry=1.0, nzdiag=11, nz_A_plus_AT=32,
    Ndense=0, memory=1096.0, Ncmpa=0, Lnz=19, Ndiv=19, Nmultsubs_LDL=29,
    Nmultsubs_LU=39, dmax=4)
```

amd_default_control

sksparse.amd.**amd_default_control**()

Get the default control parameters for AMD.

Returns

control (*dict*) – A dictionary containing the default control parameters for AMD.

The keys are:

- 'dense_thresh': Threshold for considering a row/column dense. Rows or columns with more than `max(dense_thresh * sqrt(N), 16)` entries are permuted to the end of the matrix.
- 'aggressive': Whether to use aggressive absorption.

Added in version 0.5.0.

Exceptions and Warnings

AMDError	Base class for AMD-related errors.
AMDInvalidMatrixError	Raised when the input matrix is invalid for AMD.
AMDMemoryError	Raised when AMD runs out of memory.

AMDError

exception sksparse.amd.**AMDError**

Bases: Exception

Base class for AMD-related errors.

AMDInvalidMatrixError

exception sksparse.amd.**AMDInvalidMatrixError**

Bases: *AMDError*, ValueError

Raised when the input matrix is invalid for AMD.

AMDMemoryError

exception sksparse.amd.**AMDMemoryError**

Bases: *AMDError*, MemoryError

Raised when AMD runs out of memory.

References

- SuiteSparse homepage: https://people.engr.tamu.edu/davis/suitesparse.html
- SuiteSparse AMD: https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/AMD
- AMD Algorithm Publication: Amestoy, P. R., Davis, T. A., & Duff, I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4), 886-905.

4.3.2 Block Triangular Form (BTF) (*sksparse.btf*)

Added in version 0.5.0.

Python interface to the Block Triangular Format (BTF) library.

Interface

maxtrans(A)	Compute the maximum transversal of a sparse matrix.
strongcomp(A[, q])	Compute the strongly connected components of a directed graph.
btf(A)	Permute the square sparse matrix into Block Triangular Form (BTF).
btf_q_permutation(q)	Convert a raw BTF column permutation vector to a valid permutation.

maxtrans

sksparse.btf.**maxtrans**(*A*)

Compute the maximum transversal of a sparse matrix.

This function finds a permutation of the columns of a sparse matrix so that it has a zero-free diagonal, if possible[1].

Parameters

A (*(M, N) {array-like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format.

Returns

jmatch (*(M,) ndarray*) – Array containing the maximum transversal.

Adapted from the BTF maxtrans documentation[1]:

> The output is an array `jmatch` of size `N`. If row `i` is matched with column `j`, then `A[i, j]` is nonzero, and then `jmatch[i] = j`. If the matrix is structurally nonsingular, all entries in the `jmatch` array are unique, and `jmatch` can be viewed as a column permutation if *A* is square. That is, column *k* of the original matrix becomes column `jmatch[k]` of the permuted matrix.
>
> If row `i` is not matched with any column, then `jmatch[i] = -1`.

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array
>>> from sksparse.btf import maxtrans
>>> # Create a non-symmetric matrix
>>> N = 11
>>> rng = np.random.default_rng(56)
>>> A = random_array((N, N - 3), density=0.5, format='csc', rng=rng)
>>> jmatch = maxtrans(A)
>>> jmatch
array([ 0,  2,  1,  3,  4,  5,  7, -1,  6, -1, -1], dtype=int32)
```

strongcomp

sksparse.btf.**strongcomp**(*A*, *q=None*)

Compute the strongly connected components of a directed graph.

This function finds a symmetric permutation of a sparse matrix so that `A[p][:, p]` is block upper triangular form[1].

Parameters

- **A** (*(N, N) {array-like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. Must be square.
- **q** (*(N,) ndarray of int, optional*) – A permutation vector. If provided, find the strongly connected components of `A[:, qin]`.

[1] BTF maxtrans header file: https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/BTF/Include/btf.h
[1] BTF strongcomp header file: https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/BTF/Include/btf.h

Returns

- **p** (*(N,) ndarray of int*) – The permutation vector such that `A[p][:, p]` is in block upper triangular form, unless `q` is provided (see below).
- **q** (*(N,) ndarray of int, optional*) – If `q` is provided on input, `A[p][:, q]` is in block upper triangular form.
- **r** (*(Nb+1,) ndarray of int*) – The array of indices of the start of each block in the permuted matrix. Block `b` is in rows/columns `r[b]` to `r[b+1] - 1`. The number of blocks is `len(r) - 1`.

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array, block_diag
>>> from sksparse.btf import strongcomp
>>> # Create a matrix with at least 2 strongly connected components
>>> M, N = 4, 7
>>> rng = np.random.default_rng(56)
>>> A0 = random_array((M, M), density=0.5, rng=rng)
>>> A1 = random_array((N, N), density=0.5, rng=rng)
>>> A = block_diag((A0, A1), format='csc')
>>> # The first M rows/columns are ordered together, then the last N
>>> p, r = strongcomp(A)
array([ 0,  1,  3,  2, 10,  4,  5,  6,  7,  8,  9], dtype=int32)
>>> r
array([ 0,  3,  4,  5, 11], dtype=int32)
```

btf

`sksparse.btf.`**`btf`**(*A*)

Permute the square sparse matrix into Block Triangular Form (BTF).

This function finds a permutation of a sparse matrix so that *PAQ* (`A[p][:, q]`) is block upper triangular form with a zero-free diagonal, or with a maximum number of nonzeros on the diagonal if a zero-free permutation does not exist[1].

Parameters

A (*(N, N) {array-like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. Must be square.

Returns

- **p** (*(N,) ndarray of int*) – The row permutation vector such that `A[p][:, q]` is in block upper triangular form.
- **q** (*(N,) ndarray of int*) – The column permutation vector. If `A` is structurally nonsingular, `A[p][:, q]` has a zero-free diagonal. If `A` is structurally singular, `q` will contain negative entries. The permuted matrix is `A[p][:, abs(q)]`. If `q[k] < 0`, then `PAQ[k, k]` is zero.

[1] BTF header file: https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/BTF/Include/btf.h

- **r** (*(Nb+1,) ndarray of int*) – The array of indices of the start of each block in the permuted matrix. Block b is in rows/columns `r[b]` to `r[b+1] - 1`. The number of blocks is `len(r) - 1`.

Notes

Adapted from the BTF documentation[Page 47, 1]:

> The function finds a maximum matching (or perhaps a limited matching if the work is limited), via the *maxtrans()* function. If a complete matching is not found, *btf()* completes the permutation, but flags the columns of `A[p][:, q]` to denote which columns are not matched. If the matrix is structurally rank deficient, some of the entries on the diagonal of the permuted matrix will be zero.

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array, block_diag
>>> from sksparse.btf import btf
>>> # Create a matrix with at least 2 strongly connected components
>>> M, N = 4, 7
>>> rng = np.random.default_rng(56)
>>> A0 = random_array((M, M), density=0.5, rng=rng)
>>> A1 = random_array((N, N), density=0.5, rng=rng)
>>> A = block_diag((A0, A1), format='csc')
>>> # The first M rows/columns are ordered together, then the last N
>>> p, q, r = btf(A)
>>> p
array([ 0,  3,  1,  2, 10,  4,  5,  6,  7,  8,  9], dtype=int32)
>>> q
array([ 0,  1,  2, -5, 10,  6,  5,  7,  8,  4,  9], dtype=int32)
>>> r
array([ 0,  2,  3,  4,  5, 11,  9, 10,  0,  0,  0,  0], dtype=int32)
```

btf_q_permutation

`sksparse.btf.`**`btf_q_permutation`**`(q)`

Convert a raw BTF column permutation vector to a valid permutation.

Parameters

q (*(N,) ndarray of int*) – The raw BTF column permutation vector. Contains negative entries for unmatched columns.

Returns

q_perm (*(N,) ndarray of int*) – The valid BTF column permutation vector. Contains only non-negative entries, where unmatched columns are replaced with their shifted absolute values.

Notes

In C, the values of `q` are converted using `j = BTF_UNFLIP(Q[k])`, which is a macro for:

```
j = (Q[k] < 0) ? -Q[k] - 2 : Q[k]
```

This function is a Python equivalent of that macro.

Added in version 0.5.0.

Examples

```
>>> import numpy as np
>>> from sksparse.btf import btf_q_permutation
>>> q = np.array([0, 1, 2, -5, 10, 6, 5, 7, 8, 4, 9], dtype=np.int32)
>>> btf_q_permutation(q)
array([ 0,  1,  2,  3, 10,  6,  5,  7,  8,  4,  9], dtype=int32)
```

References

- SuiteSparse homepage
- SuiteSparse BTF
- Duff, Iain. "On Algorithms for Obtaining a Maximum Transversal", *ACM Trans. Mathematical Software*, vol 7, no. 1, pp. 315-330.
- "Algorithm 575: Permutations for a Zero-Free Diagonal", *ACM Trans. Mathematical Software*, vol 7, no. 1, pp. 387-390. Algorithm 575 is MC21A in the Harwell Subroutine Library.

4.3.3 Constrained Approximate Minimum Degree (CAMD) Ordering (*sksparse.camd*)

Added in version 0.5.0.

Python interface to the Constrained Approximate Minimum Degree (CAMD) ordering algorithm.

Interface

CAMDInfo(status, N, nz, symmetry, nzdiag, ...)	Information statistics returned by the CAMD algorithm.
camd(A[, constraints, dense_thresh, ...])	Compute the approximate minimum degree ordering of a sparse matrix.
camd_default_control()	Get the default control parameters for CAMD.

CAMDInfo

class sksparse.camd.**CAMDInfo**(*status: int*, *N: int*, *nz: int*, *symmetry: float*, *nzdiag: int*, *nz_A_plus_AT: int*, *Ndense: int*, *memory: float*, *Ncmpa: int*, *Lnz: int*, *Ndiv: int*, *Nmultsubs_LDL: int*, *Nmultsubs_LU: int*, *dmax: int*)

Information statistics returned by the CAMD algorithm.

This class wraps the contents of the `Info` array output by `camd_order()` into a Python dataclass.

status

Return status:

- 0 = OK,
- 1 = OK but jumbled,
- -1 = out of memory,
- -2 = invalid matrix.

Type
int

`N`

Number of rows and columns of the input matrix `A`.

Type
int

`nz`

Number of nonzeros in the input matrix `A`.

Type
int

`symmetry`

Symmetry of pattern of `A`. The symmetry is the number of "matched" off-diagonal entries divided by the total number of off-diagonal entries. An entry `A[i, j]` is matched if `A[j, i]` is also an entry, for any pair `[i, j]` where `i != j`. In python code:

```
S = A.astype(bool)
B = sparse.tril(S, -1) + sparse.triu(S, 1)
symmetry = (B * B.T).nnz / B.nnz
```

Type
`float` $\in [0, 1]$

`nzdiag`

Number of entries on the diagonal of `A`.

Type
int

`nz_A_plus_AT`

Number of nonzeros in `A + A.T` (excluding diagonal). If `A` is perfectly symmetric (`symmetry = 1`), with a fully non-zero diagonal, then `nz_A_plus_AT = nz - N` (the smallest possible value). If `A` is perfectly unsymmetric (`symmetry = 0`, for an upper triangular matrix, *e.g.*) with no diagonal, then `nz_A_plus_AT = 2 * nz` (the largest possible value).

Type
int

`Ndense`

Number of dense rows/columns ignored during ordering. These rows/columns are placed last in the output order `p`.

Type
int

`memory`

Memory used, in bytes. This is equal to: `(1.2 * nz_A_plus_AT + 9 * N) * sizeof(int)`. This coefficient is at most `2.4 * nz + 9 * N`. This accounting excludes the size of the input arguments `Ap`, `Ai`, and `p`, which have a total size of `nz + 2 * N + 1` integers.

Type
float

Ncmpa

Number of components in the matrix (excluding dense rows/columns).

Type
int

Lnz

Number of nonzeros in the Cholesky factor `L` of `A`, excluding the diagonal. This is a slight upper bound because of the approximate degree algorithm. It is a rough upper bound if there are many dense rows/columns. The remaining statistics are also slight or rough upper bounds for the same reason.

Type
int

Ndiv

Number of division operations for LU or Cholesky factorization of the permuted matrix `A[p][:, p]`.

Type
int

Nmultsubs_LDL

Number of multiply-subtract pairs for `LDL.T` factorization.

Type
int

Nmultsubs_LU

Number of multiply-subtract pairs for LU factorization, assuming that no numerical pivoting is required.

Type
int

dmax

Maximum number of nonzeros in any column of `L`, including the diagonal.

Type
int

Notes

Field descriptions are adapted from SuiteSparse `camd.h`[1].

Added in version 0.5.0.

References

Methods

__init__
from_array

__init__

CAMDInfo.**__init__**(*status: int*, *N: int*, *nz: int*, *symmetry: float*, *nzdiag: int*, *nz_A_plus_AT: int*, *Ndense: int*, *memory: float*, *Ncmpa: int*, *Lnz: int*, *Ndiv: int*, *Nmultsubs_LDL: int*, *Nmultsubs_LU: int*, *dmax: int*) → None

[1] `camd.h` - SuiteSparse CAMD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CAMD/Include/camd.h

from_array

classmethod CAMDInfo.**from_array**(*info: ndarray*) → *CAMDInfo*

camd

sksparse.camd.**camd**(*A*, *constraints=None*, *dense_thresh=None*, *aggressive=None*, *return_info=False*)

Compute the approximate minimum degree ordering of a sparse matrix.

Adapted from the SuiteSparse *camd.h* documentation[0]:

> CAMD finds a fill-reducing ordering of a sparse matrix `A`, using the approximate minimum degree algorithm. The output is a permutation vector `p` such that the Cholesky factor of `A[p][:, p]` has fewer nonzeros than the Cholesky factor of `A`. If `A` is not symmetric, the algorithm computes an ordering of `A + A.T`.

For more details on the entire package, see the SuiteSparse homepage[1] and Github repository[2].

Parameters

- **A** (*(N, N) array_like or sparse matrix*) – A square matrix in CSC format or convertible to CSC.
- **constraints** (*(N,) array_like, optional*) – A 1D array of constraints for the ordering. Each node *i* in the graph of *A* has a constraint, `constraints[i]`, in the range [0, N-1]. All nodes with `constraints[i] = 0` are ordered first, followed by nodes with *C(i) = 1*, and so on. Thus, `constraints[p]` is monotonically non-decreasing. If None, no constraints are applied, and the ordering will be similar to *amd()*, except that the post-ordering is different.
- **dense_thresh** (*float, optional*) – Threshold number of entries for considering a row/column dense. If None, use the default value from CAMD. The default value is 10.

  Adapted from the SuiteSparse *camd.h* documentation[0]:

  > A dense row/column in `A + A.T` can cause CAMD to spend a lot of time in ordering the matrix. If `dense_thresh >= 0`, rows/columns with more than `max(dense_thresh * sqrt(N), 16)` entries are ignored during the ordering, and placed last in the output order. The default value of `dense_thresh` is 10. If negative, no rows/columns are treated as "dense". Rows/columns with 16 or fewer off-diagonal entries are never considered "dense".
- **aggressive** (*bool, optional*) – If True, use aggressive absorption. If None, uses the default value from CAMD. The default value is True.

  Adapted from the SuiteSparse *camd.h* documentation[0]:

  > Controls whether or not to use aggressive absorption, in which a prior element is absorbed into the current element if is a subset of the current element, even if it is not adjacent to the current pivot element (refer to Amestoy, Davis, & Duff, 1996, for more details). The default value is `True`, which means to perform aggressive absorption. This nearly always leads to a better ordering (because the approximate degrees are more accurate) and a lower execution time. There are cases where it can lead to a slightly worse ordering, however.
- **return_info** (*bool, optional*) – If True, returns additional information about the ordering process. Default is False.

Returns

[0] *camd.h* - Source header file from SuiteSparse. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CAMD/Include/camd.h
[1] SuiteSparse homepage. https://people.engr.tamu.edu/davis/suitesparse.html
[2] SuiteSparse GitHub repository. https://github.com/DrTimothyAldenDavis/SuiteSparse

- **p** (*ndarray*) – The permutation vector such that the Cholesky factor of `A[p][:, p]` has fewer nonzeros than the Cholesky factor of `A`.
- **info** (*ndarray, optional*) – Additional information about the ordering process, returned if `return_info` is True. Contains various statistics and status codes.

Raises

- `SparseEfficiencyWarning` – If the input matrix is not in CSC format, a warning is raised and the matrix is converted to CSC format.
- `ValueError` – If the input matrix is not square or cannot be converted to CSC format.
- *`CAMDInvalidMatrixError`* – If the input matrix is invalid for CAMD, such as having unsupported data types or formats.
- *`CAMDMemoryError`* – If the CAMD algorithm runs out of memory during execution.

See also

amd, *colamd*, *ccolamd*

Notes

This function wraps the CAMD (Approximate Minimum Degree) algorithm from the SuiteSparse by Timothy A. Davis. For details, see the SuiteSparse repository[Page 52, 2].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.camd import camd
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(565656)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> # Constrain the first K nodes to be ordered first
>>> K = 4
>>> C = np.full(N, K)
>>> C[:K] = np.arange(K)  # constrained nodes
>>> p, info = camd(A, constraints=C, return_info=True)
>>> p
array([ 0,  1,  2,  3,  8,  5,  6,  9,  4, 10,  7])
>>> info
CAMDInfo(status=0, N=11, nz=43, symmetry=1.0, nzdiag=11, nz_A_plus_AT=32,
    Ndense=0, memory=1248.0, Ncmpa=0, Lnz=19, Ndiv=19, Nmultsubs_LDL=29,
    Nmultsubs_LU=39, dmax=4)
```

camd_default_control

sksparse.camd.**camd_default_control**()

Get the default control parameters for CAMD.

Returns

control (*dict*) – A dictionary containing the default control parameters for CAMD.

The keys are:

- 'dense_thresh': Threshold for considering a row/column dense. Rows or columns with more than `max(dense_thresh * sqrt(N), 16)` entries are permuted to the end of the matrix.
- 'aggressive': Whether to use aggressive absorption.

Added in version 0.5.0.

Exceptions and Warnings

CAMDError	Base class for CAMD-related errors.
CAMDInvalidMatrixError	Raised when the input matrix is invalid for CAMD.
CAMDMemoryError	Raised when CAMD runs out of memory.

CAMDError

exception sksparse.camd.**CAMDError**

Bases: Exception

Base class for CAMD-related errors.

CAMDInvalidMatrixError

exception sksparse.camd.**CAMDInvalidMatrixError**

Bases: *CAMDError*, ValueError

Raised when the input matrix is invalid for CAMD.

CAMDMemoryError

exception sksparse.camd.**CAMDMemoryError**

Bases: *CAMDError*, MemoryError

Raised when CAMD runs out of memory.

References

- SuiteSparse homepage
- SuiteSparse CAMD
- AMD Algorithm Publication: Amestoy, P. R., Davis, T. A., & Duff, I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4), 886-905.

4.3.4 Constrained Column Approximate Minimum Degree (CCOLAMD) Ordering (*sksparse.ccolamd*)

Added in version 0.5.0.

Python interface to the Constrained Column Approximate Minimum Degree (CCOLAMD) ordering algorithm.

Interface

ccolamd(A[, constraints, dense_row_thresh, ...])	Compute the column approximate minimum degree ordering of a sparse matrix.
csymamd(A[, constraints, dense_row_thresh, ...])	Compute the column approximate minimum degree ordering of a sparse symmetric matrix.
ccolamd_get_defaults()	Get the default knobs for CCOLAMD.

ccolamd

sksparse.ccolamd.**ccolamd**(*A*, *constraints=None*, *dense_row_thresh=None*, *dense_col_thresh=None*, *aggressive=None*, *opt_lu=None*, *return_info=False*)

Compute the column approximate minimum degree ordering of a sparse matrix.

Adapted from the CCOLAMD documentation[1]:

> This function computes a column ordering for a sparse matrix *A* that is appropriate for LU factorization of symmetric or unsymmetric matrices, QR factorization, least squares, interior point methods for linear programming problems, and other related problems.
>
> CCOLAMD computes a permutation *Q* such that the Cholesky factorization of $(AQ)^\top(AQ)$ has less fill-in and requires fewer floating point operations than $A^\top A$. This also provides a good ordering for sparse partial pivoting methods, $P(AQ) = LU$, where *Q* is computed prior to numerical factorization, and *P* is computed during numerical factorization via conventional partial pivoting with row interchanges.
>
> **Parameters**
>
> - **A** (*(M, N) {array_like, sparse matrix}*) – The input matrix for which to compute the column ordering. Must be 2D and convertible to CSC format. Need not be square.
> - **contraints** (*(N,) array_like, optional*) – A 1D array of constraints for the ordering. Each column *i* in *A* has a constraint, `constraints[i]`, in the range [0, N-1]. All columns with `constraints[i] = 0` are ordered first, followed by nodes with *C(i) = 1*, and so on. Thus, `constraints[p]` is monotonically non-decreasing. If None, no constraints are applied, and the ordering will be similar to *colamd()*, except that the default values of `dense_row_thresh`, `dense_col_thresh`, and `aggressive` may differ.
> - **dense_row_thresh, dense_col_thresh** (*float, optional*) – Threshold for considering a row/column dense. If None, use the default value from CCOLAMD. The default value is 10. The actual number of entries in a row/column is to be considered "dense" is `max(dense_row_thresh * sqrt(M), 16)` where `M` is the number of rows (or `N` for columns). Dense rows/columns are ignored during ordering and moved to the end of the matrix.
> - **aggressive** (*bool, optional*) – If True, use aggressive absorption. If None, uses the default value from CCOLAMD. The default value is True.

[1] `ccolamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CCOLAMD/Source/ccolamd.c

- **opt_lu** (*{'lu', 'cholesky'}, optional*) – If 'lu', the ordering is optimized for LU factorization of A. If 'cholesky', the ordering is optimized for Cholesky factorization of $A^\top A$. If None, uses the default value from CCOLAMD, which is 'cholesky'.

Returns

- **q** ((N,) ndarray) – The permutation vector.
- **stats** (*CCOLAMDStats*, optional) – If `return_info` is True, returns an object containing statistics about the ordering.

See also

csymamd, *colamd*, *symamd*

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array
>>> from sksparse.ccolamd import ccolamd
>>> # Create a non-symmetric matrix
>>> N = 11
>>> rng = np.random.default_rng(56)
>>> A = random_array((N, N - 3), density=0.5, format='csc', rng=rng)
>>> A.setdiag(N)  # make the diagonal non-zero
>>> # Constrain the first K nodes to be ordered first
>>> K = 4
>>> C = np.full(A.shape[1], K)
>>> C[:K] = np.arange(K)  # constrained nodes
>>> p, info = ccolamd(A, constraints=C, return_info=True)
>>> p
array([0, 1, 2, 3, 4, 7, 6, 5], dtype=int32)
>>> info
CCOLAMDStats(N_rows_ignored=0, N_cols_ignored=0, Ncmpa=0, status=0, info1=-1,
    info2=-1, info3=0)
```

csymamd

sksparse.ccolamd.**csymamd**(*A*, *constraints=None*, *dense_row_thresh=None*, *dense_col_thresh=None*, *aggressive=None*, *return_info=False*)

Compute the column approximate minimum degree ordering of a sparse symmetric matrix.

Adapted from the CCOLAMD documentation[1]:

> This function computes an approximate minimum degree ordering for Cholesky factorization of symmetric matrices.
>
> Symamd computes a permutation P of a symmetric matrix A such that the Cholesky factorization of $PAP^\top$ has less fill-in and requires fewer floating point operations than A. Symamd constructs a

[1] `ccolamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CCOLAMD/Source/ccolamd.c

matrix M such that $M^\top M$ has the same nonzero pattern of A, and then orders the columns of M using ccolamd. The column ordering of M is then returned as the row and column ordering P of A.

Parameters

A (*(N, N) array_like or sparse matrix*) – The input matrix for which to compute the column ordering. Must be 2D, square, and convertible to CSC format.

> **Note**
>
> This routine only accesses the lower triangular part of `A`, which is *assumed* to be symmetric. If it is not, the results may be incorrect or undefined.

contraints

[(N,) array_like, optional] A 1D array of constraints for the ordering. Each column *i* in *A* has a constraint, `constraints[i]`, in the range [0, N-1]. All columns with `constraints[i] = 0` are ordered first, followed by nodes with *C(i) = 1*, and so on. Thus, `constraints[p]` is monotonically non-decreasing. If None, no constraints are applied, and the ordering will be similar to *colamd()*, except that the default values of `dense_row_thresh`, `dense_col_thresh`, and `aggressive` may differ.

dense_row_thresh, dense_col_thresh

[float, optional] Threshold for considering a row/column dense. If None, use the default value from CCOLAMD. The default value is 10. The actual number of entries in a row/column is to be considered "dense" is `max(dense_row_thresh * sqrt(M), 16)` where `M` is the number of rows (or `N` for columns). Dense rows/columns are ignored during ordering and moved to the end of the matrix.

aggressive

[bool, optional] If True, use aggressive absorption. If None, uses the default value from CCOLAMD. The default value is True.

Returns

- **q** ((N,) `ndarray`) – The permutation vector.
- **stats** (*CCOLAMDStats*, optional) – If `return_info` is True, returns an object containing statistics about the ordering.

> **See also**
>
> *ccolamd*, *colamd*, *symamd*

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array
>>> from sksparse.ccolamd import csymamd
>>> # Create a non-symmetric matrix
>>> N = 11
>>> rng = np.random.default_rng(56)
>>> A = random_array((N, N - 3), density=0.5, format='csc', rng=rng)
```

(continues on next page)

(continued from previous page)

```
>>> A.setdiag(N)  # make the diagonal non-zero
>>> A = (A.T @ A).tocsc()  # make A symmetric
>>> # Constrain the first K nodes to be ordered first
>>> K = 4
>>> C = np.full(A.shape[1], K)
>>> C[:K] = np.arange(K)   # constrained nodes
>>> p, info = csymamd(A, constraints=C, return_info=True)
>>> p
array([0, 1, 2, 3, 7, 6, 5, 4], dtype=int32)
>>> info
CCOLAMDStats(N_rows_ignored=0, N_cols_ignored=0, Ncmpa=0, status=0, info1=-1,
    info2=-1, info3=0)
```

ccolamd_get_defaults

sksparse.ccolamd.**ccolamd_get_defaults**()

Get the default knobs for CCOLAMD.

Returns

knobs (*dict*) – A dictionary containing the default knobs for CCOLAMD.

The keys are:

- 'dense_row_thresh': Threshold for considering a row/column dense. Rows with more than `max(dense_row_thresh * sqrt(M), 16)` entries are permuted to the end of the matrix.
- 'dense_col_thresh': Like *dense_row_thresh*, but for columns.
- 'aggressive': Default value for the aggressive knob.

Added in version 0.5.0.

Exceptions and Warnings

CCOLAMDError	Base class for CCOLAMD errors.
CCOLAMDValueError	Raised when CCOLAMD encounters a value error.
CCOLAMDMemoryError	Raised when CCOLAMD runs out of memory.
CCOLAMDInternalError	Raised when CCOLAMD encounters an internal error.
CCOLAMDStats(N_rows_ignored, N_cols_ignored, ...)	Information statistics returned by the CCOLAMD algorithm.

CCOLAMDError

exception sksparse.ccolamd.**CCOLAMDError**

Bases: Exception

Base class for CCOLAMD errors.

CCOLAMDValueError

exception sksparse.ccolamd.**CCOLAMDValueError**

Bases: *CCOLAMDError*, ValueError

Raised when CCOLAMD encounters a value error.

CCOLAMDMemoryError

exception sksparse.ccolamd.**CCOLAMDMemoryError**

Bases: *CCOLAMDError*, MemoryError

Raised when CCOLAMD runs out of memory.

CCOLAMDInternalError

exception sksparse.ccolamd.**CCOLAMDInternalError**

Bases: *CCOLAMDError*, RuntimeError

Raised when CCOLAMD encounters an internal error.

CCOLAMDStats

class sksparse.ccolamd.**CCOLAMDStats**(*N_rows_ignored: int*, *N_cols_ignored: int*, *Ncmpa: int*, *status: int*, *info1: int*, *info2: int*, *info3: int*)

Information statistics returned by the CCOLAMD algorithm.

This class wraps the contents of the `stats` array returned by C `ccolamd()` into a Python dataclass.

N_rows_ignored

The number of dense or empty rows ignored in the ordering.

Type
int

N_cols_ignored

The number of dense or empty columns ignored in the ordering.

Type
int

Ncmpa

The number of garbage collections performed.

Type
int

status

Status code indicating the result of the CCOLAMD operation. If non-zero, `ccolamd` will throw an appropriate exception that interprets this status code.

Type
int

The following fields take on different meanings depending on the value of

``status``

info1

Value of `status`:

- 0: the highest numbered column that is unsorted or has duplicate entries.
- -3: the value of `n_row`.
- -4: the value of `n_col`.
- -5: the value of `nnz == p[n_col]`.

- -6: the value of `p[0]`.
- -7: the required `Alen` value.
- -8: the column with negative entries.
- -9: the column with a row index out of bounds.

Type
int

info2

Value of `status`:

- 0: the last seen duplicate or unsorted row index.
- -7: the actual `Alen` value.
- -9: the bad row index.

Type
int

info3

Value of `status`:

- 0: the number of duplicates or unsorted row indices.
- -9: `n_row`.

Type
int

Notes

Field descriptions are adapted from SuiteSparse `ccolamd.c`[1].

Added in version 0.5.0.

References

Methods

__init__	
from_array	Create a CCOLAMDStats instance from an array.

__init__

CCOLAMDStats.**__init__**(*N_rows_ignored: int*, *N_cols_ignored: int*, *Ncmpa: int*, *status: int*, *info1: int*, *info2: int*, *info3: int*) → None

[1] `ccolamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CCOLAMD/Source/ccolamd.c

from_array

classmethod CCOLAMDStats.**from_array**(*stats: ndarray*) → *CCOLAMDStats*

Create a CCOLAMDStats instance from an array.

References

- SuiteSparse homepage
- SuiteSparse CCOLAMD
- CCOLAMD Algorithm Publications:
  - T. A. Davis, J. R. Gilbert, S. Larimore, E. Ng, An approximate column minimum degree ordering algorithm, *ACM Transactions on Mathematical Software*, vol. 30, no. 3., pp. 353-376, 2004.
  - T. A. Davis, J. R. Gilbert, S. Larimore, E. Ng, Algorithm 836: CCOLAMD, an approximate column minimum degree ordering algorithm, *ACM Transactions on Mathematical Software*, vol. 30, no. 3., pp. 377-380, 2004.

4.3.5 Cholesky Decomposition (*sksparse.cholmod*)

Added in version 0.1.0.

Changed in version 0.5.0: Major API updates to more closely resemble the `scipy.linalg.cholesky()` dense interface, and incorporate more functions from the CHOLMOD MATLAB interface.

An interface to the SuiteSparse CHOLMOD package, which computes basic linear algebra operations for sparse, symmetric, positive-definite matrices.

Function Interface

cholesky	Compute the Cholesky factorization of a sparse matrix.
ldl	Compute the LDL factorization of a sparse matrix.
cho_solve	Solve the linear system $A\,x = b$ for x, using the Cholesky factorization.

cholesky

sksparse.cholmod.**cholesky**(*A*, *beta=0.0*, ** (Keyword-only parameters separator (PEP 3102))*, *lower=False*, *order='default'*, *sym_kind=None*, *supernodal_mode=None*)

Compute the Cholesky factorization of a sparse matrix.

This function computes the Cholesky factorization of a symmetric positive definite matrix A:

$$R^\top R = PAP^\top,$$

where R is an upper triangular matrix. Only the upper triangular part of A is used. If `lower` is True, the lower triangular factor L is returned instead, such that:

$$LL^\top = PAP^\top.$$

In this case, only the lower triangular part of A is used.

If `beta` is a scalar value, compute the factorization of:

$$PAP^\top + \beta I,$$

where I is the identity matrix.

Parameters

- **A** (*(N, N) {array_like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. The matrix must be square and symmetric positive definite. Only the upper or lower triangular part of the matrix is used, and no check is made for symmetry.
- **beta** (*float, optional*) – The scalar value to add to the diagonal of the matrix before factorization.
- **lower** (*bool, optional*) – If True, return the lower triangular factor *L*, otherwise return the upper triangular factor *R*.
- **order** (*None or str in {"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.
  - `best`: Automatically select the best ordering based on the input.
  - `metis`: Use the METIS library for graph partitioning.
  - `nesdis`: Use the NESDIS library for nested dissection.
  - `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
  - **`colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the** symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).
  - `postordered`: Use natural ordering followed by postordering.
  - `natural` or `None`: No permutation is applied (identity permutation).

  By default, methods other than `natural` will also be postordered.

  > **Warning**
  >
  > The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

- **sym_kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. No check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.
- **supernodal_mode** (*str in {"auto", "simplicial", "supernodal"}, optional*) – The type of factorization to use:
  - `auto`: Automatically select the factorization type.
  - `simplicial`: Use a simplicial factorization.
  - `supernodal`: Use a supernodal factorization.

  Note that the `simplicial` mode may be slow for large matrices.

Returns

- **R** (*csc_array*) – The triangular factor of the Cholesky decomposition. The data type will match that of `A`.
- **p** (*ndarray of int, optional*) – The permutation vector used in the factorization. Only returned if the ordering is not `None`.

Raises

CholmodNotPositiveDefiniteError – If the input matrix is not positive definite.

See also

cho_factor, *ldl*, *ldl_factor*

Notes

This function is an interface to the CHOLMOD library, which is part of the SuiteSparse collection by Timothy A. Davis. For more details, see the documentation in the header file[1].

Added in version 0.1.0.

Changed in version 0.5.0: The function now returns the matrix directly instead of a `Factor` object, and the permutation vector when an ordering method is specified.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import cholesky, cho_factor
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> L, p = cholesky(A, order='amd', lower=True)
>>> L
<Compressed Sparse Column sparse array of dtype 'float64'
        with 30 stored elements and shape (11, 11)>
>>> p
array([ 4,  8,  6,  0,  3,  5,  1,  2,  9, 10,  7])
>>> f = cho_factor(A, order='amd', lower=True)
>>> f
CholeskyFactor(N=11, nnz=30, is_ll=True, is_super=False, itype=np.int64,
    dtype=np.float64, order=natural)
>>> np.allclose(L.toarray(), f.get_factor().toarray(), atol=1e-15)
True
>>> np.array_equal(p, f.get_perm())
```

(continues on next page)

[1] `cholmod.h` - SuiteSparse CHOLMOD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Include/cholmod.h

(continued from previous page)

```
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

ldl

sksparse.cholmod.**ldl**(*A*, *beta=0.0*, *, *lower=True*, *order='default'*, *sym_kind=None*, *supernodal_mode=None*)

Compute the LDL factorization of a sparse matrix.

This function computes the LDL factorization of a symmetric matrix A:

$$LDL^\top = PAP^\top,$$

where L is a lower triangular matrix with unit diagonal, and D is a diagonal matrix. Only the lower triangular part of A is used. If `lower` is False, the upper triangular factor R is returned instead, such that:

$$R^\top DR = PAP^\top.$$

In this case, only the upper triangular part of A is used.

If `beta` is a scalar value, compute the factorization of:

$$PAP^\top + \beta I,$$

where I is the identity matrix.

Parameters

- **A** (*(N, N) {array_like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. The matrix must be square and symmetric positive definite. Only the upper or lower triangular part of the matrix is used, and no check is made for symmetry.
- **beta** (*float, optional*) – The scalar value to add to the diagonal of the matrix before factorization.
- **lower** (*bool, optional*) – If True, return the lower triangular factor L, otherwise return the upper triangular factor R.
- **order** (*None or str in {"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.
  - `best`: Automatically select the best ordering based on the input.
  - `metis`: Use the METIS library for graph partitioning.
  - `nesdis`: Use the NESDIS library for nested dissection.
  - `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
  - **`colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the** symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).

- `postordered`: Use natural ordering followed by postordering.
- `natural` or `None`: No permutation is applied (identity permutation).

By default, methods other than `natural` will also be postordered.

> **Warning**
>
> The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

- **sym_kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. No check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.
- **supernodal_mode** (*str in {"auto", "simplicial", "supernodal"}, optional*) – The type of factorization to use:
  - `auto`: Automatically select the factorization type.
  - `simplicial`: Use a simplicial factorization.
  - `supernodal`: Use a supernodal factorization.

  Note that the `simplicial` mode may be slow for large matrices.

Returns

- **R** (*csc_array*) – The triangular factor of the Cholesky decomposition. The data type will match that of `A`.
- **D** (*dia_array*) – The diagonal matrix D of the factorization, in sparse DIA format. The data type will match that of `A`.
- **p** (*ndarray of int, optional*) – The permutation vector used in the factorization. Only returned if the ordering is not `None`.

Raises

CholmodNotPositiveDefiniteError – If the input matrix is not positive definite.

> **See also**
>
> *ldl_factor*, *cholesky*, *cho_factor*

Notes

This function is an interface to the CHOLMOD library, which is part of the SuiteSparse collection by Timothy A. Davis. For more details, see the documentation in the header file[1].

Added in version 0.5.0.

[1] `cholmod.h` - SuiteSparse CHOLMOD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Include/cholmod.h

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import ldl, ldl_factor
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> L, D, p = ldl(A, order='amd')
>>> L
<Compressed Sparse Column sparse array of dtype 'float64'
        with 30 stored elements and shape (11, 11)>
>>> D
<DIAgonal sparse array of dtype 'float64'
        with 11 stored elements (1 diagonals) and shape (11, 11)>
>>> p
array([ 4,  8,  6,  0,  3,  5,  1,  2,  9, 10,  7])
>>> f = ldl_factor(A, order='amd')
>>> f
CholeskyFactor(N=11, nnz=30, is_ll=False, is_super=False, itype=np.int64,
    dtype=np.float64, order=amd)
>>> Lf, Df = f.get_factor()
>>> np.allclose(L.toarray(), Lf.toarray(), atol=1e-15)
True
>>> np.allclose(D.toarray(), Df.toarray(), atol=1e-15)
True
>>> np.array_equal(p, f.get_perm())
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

cho_solve

sksparse.cholmod.**cho_solve**(*A*, *b*, ***, *lower=False*, *order='default'*)

Solve the linear system $A\,x = b$ for x, using the Cholesky factorization.

The matrix A is factorized using the same parameters as in *cho_factor()*, and subject to the same positive definiteness requirements.

Parameters

- **b** (*(N,) or (N, K) ndarray or sparse matrix*) – The right-hand side vector or matrix. Must be

a type that can be safely cast to the data type of the `A` matrix. The number of rows in `b` must be equal to the size of the `A` matrix.

- **lower** (*bool, optional*) – If True, use only the lower triangular part of `A`, otherwise use the upper triangular part.
- **order** (*None or str in {{"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.
  - `best`: Automatically select the best ordering based on the input.
  - `metis`: Use the METIS library for graph partitioning.
  - `nesdis`: Use the NESDIS library for nested dissection.
  - `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
  - **`colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the** symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).
  - `postordered`: Use natural ordering followed by postordering.
  - `natural` or `None`: No permutation is applied (identity permutation).

  By default, methods other than `natural` will also be postordered.

> **Warning**
>
> The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

Returns

x (*(N,) or (N, K) ndarray or sparse matrix*) – The solution vector or matrix, returned in the same format as `b`.

Raises

`CholmodNotPositiveDefiniteError` – If the matrix `A` is exactly singular, or singular to working precision.

Notes

This function solves the linear system:

$$R^\top Rx = b,$$

where *R* is the upper triangular factor from the Cholesky factorization of *A*. The input *b* is either dense or sparse, vector or matrix.

If `order` was not `natural` when the factorization was computed, solve the system:

$$P^\top R^\top RPx = b$$

where *P* is the permutation matrix corresponding to the permutation vector. Similarly, if `lower` was True when the factorization was computed, the system solved is:

$$P^\top LL^\top Px = b.$$

This function uses the CHOLMOD library to solve the linear system. It is intended to combine the MATLAB interfaces `cholmod2.m`[1].

Added in version 0.5.0.

References

Object Interface

cho_factor	Compute the Cholesky factorization of a sparse matrix.
ldl_factor	Compute the LDL factorization of a sparse matrix.
CholeskyFactor	The main object used for creating and manipulating a Cholesky factor.

cho_factor

sksparse.cholmod.**cho_factor**(*A*, *beta=0.0*, ***, *lower=False*, *order='default'*, *sym_kind=None*, *supernodal_mode=None*)

Compute the Cholesky factorization of a sparse matrix.

This function computes the Cholesky factorization of a symmetric positive definite matrix A:

$$R^\top R = PAP^\top,$$

where R is an upper triangular matrix. Only the upper triangular part of A is used. If `lower` is True, the lower triangular factor L is returned instead, such that:

$$LL^\top = PAP^\top.$$

In this case, only the lower triangular part of A is used.

If `beta` is a scalar value, compute the factorization of:

$$PAP^\top + \beta I,$$

where I is the identity matrix.

Parameters

- **A** (*(N, N) {array_like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. The matrix must be square and symmetric positive definite. Only the upper or lower triangular part of the matrix is used, and no check is made for symmetry.
- **beta** (*float, optional*) – The scalar value to add to the diagonal of the matrix before factorization.
- **lower** (*bool, optional*) – If True, return the lower triangular factor L, otherwise return the upper triangular factor R.
- **order** (*None or str in {"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.

[1] `cholmod2.c` - CHOLMOD MATLAB interface https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/cholmod2.c

- `best`: Automatically select the best ordering based on the input.
- `metis`: Use the METIS library for graph partitioning.
- `nesdis`: Use the NESDIS library for nested dissection.
- `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
- **`colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the** symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).
- `postordered`: Use natural ordering followed by postordering.
- `natural` or `None`: No permutation is applied (identity permutation).

By default, methods other than `natural` will also be postordered.

> **Warning**
>
> The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

- **sym_kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. No check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.
- **supernodal_mode** (*str in {"auto", "simplicial", "supernodal"}, optional*) – The type of factorization to use:
  - `auto`: Automatically select the factorization type.
  - `simplicial`: Use a simplicial factorization.
  - `supernodal`: Use a supernodal factorization.

  Note that the `simplicial` mode may be slow for large matrices.

Returns

CholeskyFactor – The factorization object. Use its methods to solve linear systems and manipulate the factorization.

Raises

CholmodNotPositiveDefiniteError – If the input matrix is not positive definite.

> **See also**
>
> *cholesky*, *ldl*, *ldl_factor*

Notes

This function is an interface to the CHOLMOD library, which is part of the SuiteSparse collection by Timothy A. Davis. For more details, see the documentation in the header file[1].

Added in version 0.5.0.

[1] `cholmod.h` - SuiteSparse CHOLMOD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Include/cholmod.h

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import cholesky, cho_factor
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> L, p = cholesky(A, order='amd', lower=True)
>>> L
<Compressed Sparse Column sparse array of dtype 'float64'
        with 30 stored elements and shape (11, 11)>
>>> p
array([ 4,  8,  6,  0,  3,  5,  1,  2,  9, 10,  7])
>>> f = cho_factor(A, order='amd', lower=True)
>>> f
CholeskyFactor(N=11, nnz=30, is_ll=True, is_super=False, itype=np.int64,
    dtype=np.float64, order=natural)
>>> np.allclose(L.toarray(), f.get_factor().toarray(), atol=1e-15)
True
>>> np.array_equal(p, f.get_perm())
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

ldl_factor

sksparse.cholmod.**ldl_factor**(*A*, *beta=0.0*, ***, *lower=True*, *order='default'*, *sym_kind=None*, *supernodal_mode=None*)

Compute the LDL factorization of a sparse matrix.

This function computes the LDL factorization of a symmetric matrix A:

$$LDL^\top = PAP^\top,$$

where L is a lower triangular matrix with unit diagonal, and D is a diagonal matrix. Only the lower triangular part of A is used. If `lower` is False, the upper triangular factor R is returned instead, such that:

$$R^\top DR = PAP^\top.$$

In this case, only the upper triangular part of A is used.

If `beta` is a scalar value, compute the factorization of:

$$PAP^\top + \beta I,$$

where I is the identity matrix.

Parameters

- **A** (*(N, N) {array_like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. The matrix must be square and symmetric positive definite. Only the upper or lower triangular part of the matrix is used, and no check is made for symmetry.
- **beta** (*float, optional*) – The scalar value to add to the diagonal of the matrix before factorization.
- **lower** (*bool, optional*) – If True, return the lower triangular factor *L*, otherwise return the upper triangular factor *R*.
- **order** (*None or str in {"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.
  - `best`: Automatically select the best ordering based on the input.
  - `metis`: Use the METIS library for graph partitioning.
  - `nesdis`: Use the NESDIS library for nested dissection.
  - `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
  - **`colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the** symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).
  - `postordered`: Use natural ordering followed by postordering.
  - `natural` or `None`: No permutation is applied (identity permutation).

  By default, methods other than `natural` will also be postordered.

  > **Warning**
  >
  > The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

- **sym_kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. No check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.
- **supernodal_mode** (*str in {"auto", "simplicial", "supernodal"}, optional*) – The type of factorization to use:
  - `auto`: Automatically select the factorization type.
  - `simplicial`: Use a simplicial factorization.
  - `supernodal`: Use a supernodal factorization.

Note that the `simplicial` mode may be slow for large matrices.

Returns

CholeskyFactor – The factorization object. Use its methods to solve linear systems and manipulate the factorization.

Raises

CholmodNotPositiveDefiniteError – If the input matrix is not positive definite.

See also

ldl, *cholesky*, *cho_factor*

Notes

This function is an interface to the CHOLMOD library, which is part of the SuiteSparse collection by Timothy A. Davis. For more details, see the documentation in the header file[1].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import ldl, ldl_factor
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> L, D, p = ldl(A, order='amd')
>>> L
<Compressed Sparse Column sparse array of dtype 'float64'
        with 30 stored elements and shape (11, 11)>
>>> D
<DIAgonal sparse array of dtype 'float64'
        with 11 stored elements (1 diagonals) and shape (11, 11)>
>>> p
array([ 4,  8,  6,  0,  3,  5,  1,  2,  9, 10,  7])
>>> f = ldl_factor(A, order='amd')
>>> f
CholeskyFactor(N=11, nnz=30, is_ll=False, is_super=False, itype=np.int64,
    dtype=np.float64, order=amd)
>>> Lf, Df = f.get_factor()
>>> np.allclose(L.toarray(), Lf.toarray(), atol=1e-15)
```

(continues on next page)

[1] `cholmod.h` - SuiteSparse CHOLMOD header file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Include/cholmod.h

(continued from previous page)

```
True
>>> np.allclose(D.toarray(), Df.toarray(), atol=1e-15)
True
>>> np.array_equal(p, f.get_perm())
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

CholeskyFactor

class sksparse.cholmod.**CholeskyFactor**

The main object used for creating and manipulating a Cholesky factor.

The constructor computes the symbolic analysis of the matrix and determines a fill-reducing ordering (if `order` is not `None` or `"natural"`) such that:

$$LL^\top = PAP^\top.$$

The numeric factorization is not computed until *factorize()* is called.

Parameters

- **A** (*(N, N) array_like or sparse array*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format.
- **sym_kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper or lower triangular part of the matrix is used (depending on `lower`), and no check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.

  The resulting matrix must be square and symmetric positive definite.
- **supernodal_mode** (*str in {"auto", "simplicial", "supernodal"}, optional*) – The type of factorization to use:
  - `auto`: Automatically select the factorization type.
  - `simplicial`: Use a simplicial factorization.
  - `supernodal`: Use a supernodal factorization.

  Default is `auto`. This mode also applies to any subsequent calls to *factorize()*. Note that the `simplicial` mode may be slow for large matrices.
- **lower** (*bool, optional*) – If True, use the lower triangular part of `A`.
- **order** (*str in {"default", "best", "natural", "metis", "nesdis", "amd", "colamd", "postordered"}, optional*) – The permutation algorithm to use for the factorization. Options are:
  - `default`: Use the default method, which first tries AMD, then METIS.
  - `best`: Automatically select the best ordering based on the input.

- `metis`: Use the METIS library for graph partitioning.
- `nesdis`: Use the NESDIS library for nested dissection.
- `amd`: Use the Approximate Minimum Degree (AMD) algorithm.
- `colamd`: Use the Approximate Minimum Degree (AMD) algorithm for the symmetric case, or the COLAMD algorithm for the unsymmetric case ($AA^\top$ or $A^\top A$).
- `postordered`: Use natural ordering followed by postordering.
- `natural` or `None`: No permutation is applied (identity permutation).

By default, methods other than `natural` will also be postordered.

> **Warning**
>
> The ordering method `best` may be quite slow for large matrices, but if the factorization is reused many times, it can be worth it.

N

The number of rows and columns in the factor.

Type
int

is_ll

Whether the factor is in `LL.T` form (True) or `LDL.T` form (False).

Type
bool

is_super

Whether the factor is in supernodal (True) or simplicial (False) format.

Type
bool

itype

The integer type used for indices and indptr in the factor.

Type
`numpy.int32` or `numpy.int64`

dtype

The data type used for numerical values in the factor.

Type
numpy.dtype

sym_kind

The symmetry kind used for the factorization.

Type
str

rcond

A rough estimate of the reciprocal of the condition number of the matrix, defined as `(L.diagonal().min() / L.diagonal().max())**2` for an `LL.T` factorization. Estimated during the numeric factorization. If *`factorize()`* has not yet been called, this value is `-1.0`.

Type
float

colcount

The number of nonzeros in each column of the factor.

Type
(N,) numpy.ndarray of int

nnz

The number of nonzeros in the factor.

Type
int

order

The ordering method used for the factorization. If an unknown ordering was used, returns the integer value.

Type
str or int

perm

A read-only view of the permutation vector used for the factorization.

Type
(N,) numpy.ndarray of int

factor

A view of the the Cholesky factor in Compressed Sparse Column (CSC) format. If `self.is_ll`, the returned matrix is lower triangular. Otherwise, the matrix view contains the lower triangular and the diagonal factors combined.

Note

The view is always in lower triangular form, even if the factor was created using `lower=False`. To get the upper triangular factor, use *get_factor* with `lower=False`. To get the split L and D factors, use *get_factor* with `kind="LDL"`.

Warning

The returned matrix is a view on the internal data of the CHOLMOD factor. It will be modified if the factor is modified (*e.g.*, by calling *factorize()*). To get a copy, use *get_factor()*.

Type
csc_array

L

The lower triangular factor in Compressed Sparse Column (CSC) format.

Type
csc_array

R

The upper triangular factor in Compressed Sparse Column (CSC) format.

Type
csc_array

D

A view of the diagonal factor in DIAgonal format. If `self.is_ll`, this is the identity matrix.

Type
dia_array

Raises
CholmodNotPositiveDefiniteError – If the input matrix is structurally singular (*e.g.*, if it is the zero matrix). The input *may* be numerically indefinite, but this property is not checked until *factorize()* is called.

See also

cholesky, *ldl*, *cho_factor*, *ldl_factor*

Notes

The symbolic analysis follows that of the SuiteSparse CHOLMOD `analyze` MATLAB function[1].

Warning

Calling `CholeskyFactor.__new__(CholeskyFactor)` will leave the object in an "unsafe" state, since the internal CHOLMOD structures will not be initialized. Always use the constructor `CholeskyFactor(...)` to create a new object.

Added in version 0.1.0.

Changed in version 0.5.0: Refactored to `CholeskyFactor` from just `Factor`. Now incorporates `Common` factor into the object. Major API changes.

References

Methods

__init__	
change_factor	Change the type of factorization used by the object.
copy	Return a copy of the CholeskyFactor object.
det	Compute the determinant of the matrix from its Cholesky factorization.
downdate	Multiple-rank downdate of a sparse LDL factorization.
factorize	Compute the numerical Cholesky factorization of a sparse matrix.
get_factor	Return a copy of the Cholesky factor in the specified format.
get_perm	Return a copy of the permutation vector used in the factorization.

continues on next page

[1] `analyze.c` - CHOLMOD MATLAB analyze function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/analyze.c

Table 13 – continued from previous page

inv	Compute the inverse of the matrix from its Cholesky factorization.
logdet	Compute the (natural) log-determinant of the matrix from its Cholesky factorization.
resymbol	Recompute the symbolic Cholesky factorization of a sparse matrix.
rowadd	Add a row to a sparse LDL factorization.
rowdel	Delete a row from a sparse LDL factorization.
slogdet	Compute the sign and (natural) log-determinant of the matrix from its Cholesky factorization.
solve	Solve the linear system $Ax = b$ for x, using the factorization.
update	Multiple-rank update of a sparse LDL factorization.

__init__

CholeskyFactor.**__init__**(**args*, ***kwargs*)

change_factor

CholeskyFactor.**change_factor**(*kind=None*)

Change the type of factorization used by the object.

This method changes the type of factorization used by the object between `LL.T` and `LDL.T`. The symbolic analysis is reused, so this method does not require refactorization of the matrix.

Parameters

kind (*str in {'LL', 'LDL'}, optional*) – The type of factorization to use. If `LL`, use the Cholesky factor L such that $LL^\top = PAP^\top$. If `LDL`, use the combined LD factor such that $LDL^\top = PAP^\top$. Default is None, which switches to the other type of factorization.

See also

factor, *get_factor*

copy

CholeskyFactor.**copy**()

Return a copy of the CholeskyFactor object.

This method creates a deep copy of the CholeskyFactor object, including the CHOLMOD common struct and the factor itself.

This method does not copy *all* of the underlying *cholmod_common* struct, only the parts that are necessary for using the factor.

Returns

CholeskyFactor – A deep copy of the CholeskyFactor object.

det

`CholeskyFactor.det()`

Compute the determinant of the matrix from its Cholesky factorization.

Added in version 0.2.

> **Warning**
>
> This function may overflow or underflow for large matrices. Use *logdet()* or *slogdet()* instead.

Returns

det (*float*) – The determinant of the matrix A that was factorized.

> **See also**
>
> *logdet*, *slogdet*, `numpy.linalg.det`, `numpy.linalg.slogdet`, `scipy.linalg.det`

downdate

`CholeskyFactor.downdate(C)`

Multiple-rank downdate of a sparse LDL factorization.

Compute a downdate to the factorization of a sparse matrix A[1]:

$$L'D'L'^{\top} = P(A - CC^{\top})P^{\top}$$

where L is a lower triangular matrix with unit diagonal, and D is a diagonal matrix. The input `C` is a sparse matrix representing the downdate to the factorization. The fill-reducing permutation is *not* recomputed from the original A.

Parameters

C (*(N, K) csc_array*) – The sparse matrix representing the rank-k update or downdate to the matrix.

Returns

CholeskyFactor – The current object, for method chaining.

References

Added in version 0.5.0.

factorize

`CholeskyFactor.factorize(A, ldl=None, beta=0.0)`

Compute the numerical Cholesky factorization of a sparse matrix.

This method computes the numerical values of $PAP^{\top} = R^{\top}R$ or $PAP^{\top} = LL^{\top}$ decomposition of a Hermitian positive-definite matrix A, with fill-reducing permutation P.

Parameters

[1] `cholmod_updown.c` - CHOLMOD up/downdate function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Modify/cholmod_updown.c

- **A** (*(N, N) {array_like, sparse array}*) – An array convertible to a sparse matrix in Compressed Sparse Column (CSC) format. The matrix must be square and symmetric positive definite. Only the upper or lower triangular part of the matrix is used, and no check is made for symmetry. This matrix can be numericaly different from the matrix used to initialize the `CholeskyFactor` object, but it must have the same sparsity pattern.
- **ldl** (*bool, optional*) – If True, compute the LDL factorization instead of the Cholesky factorization. Default is None, which uses the same type of factorization as the previous call to `factorize()`, or False if this is the first call.
- **beta** (*float, optional*) – The scalar value to add to the diagonal of the matrix before factorization. Default is 0.

Notes

If `ldl=False`, this function computes the Cholesky factorization of a symmetric positive definite matrix A:

$$R^\top R = PAP^\top,$$

where R is an upper triangular matrix. Only the upper triangular part of A is used. If `self.is_lower` is True, the lower triangular factor L is computed instead, such that:

$$LL^\top = PAP^\top.$$

In this case, only the lower triangular part of A is used.

If `ldl=True`, compute the factorization:

$$R^\top DR = PAP^\top,$$

or

$$LDL^\top = PAP^\top,$$

respectively.

If `beta` is not None, the factorization is computed for the matrix:

$$PAP^\top + \beta I.$$

Note that if the `CholeskyFactor` was initialized with `sym_kind` equal to `"row"` or `"col"`, the factorization is computed for $PAA^\top P^\top$ or $PA^\top AP^\top$, respectively. Similarly, `beta` is added to the diagonal of these matrices.

get_factor

`CholeskyFactor.get_factor`(*kind=None*, *lower=None*)

Return a copy of the Cholesky factor in the specified format.

Parameters

- **kind** (*None or str in {'LL', 'LDL'}, optional*) – The type of factor to return. If `LL`, return the Cholesky factor L such that $LL^\top = PAP^\top$. If `LDL`, return the combined LD factor such that $LDL^\top = PAP^\top$. Default is None, which uses the kind with which `factorize` was called.
- **lower** (*None or bool, optional*) – If True, return the lower triangular factor L. If False, return the upper triangular factor R. If None (default), return the factor in the same triangular form as it was created with `factorize`.

Returns

- **L** (*csc_array*) – The Cholesky factor in Compressed Sparse Column (CSC) format.
- **D** (*diags_array, optional*) – If `kind="LDL"`, also returns the D factor.

get_perm

`CholeskyFactor.get_perm()`

Return a copy of the permutation vector used in the factorization.

Returns

p (*ndarray*) – The permutation vector p such that $PAP^\top$ is the matrix that was factorized, where P is the permutation matrix corresponding to p, *i.e.*, `P = I[p]`.

inv

`CholeskyFactor.inv()`

Compute the inverse of the matrix from its Cholesky factorization.

Warning

For most purposes, it is better to use *solve()* instead of computing the inverse explicitly. The following two lines of code are mathematically equivalent:

```
x = f.solve(b)
x = f.inv() @ b  # DO NOT USE
```

but the first line is both faster and more numerically stable.

Returns

Ainv (*csc_array*) – The inverse of the matrix A that was factorized.

See also

`numpy.linalg.inv`, `scipy.linalg.inv`

Notes

This function computes the inverse of the matrix A from its Cholesky factorization. If the factorization is in LL^T form, the inverse is computed as:

$$A^{-1} = P^\top L^{-\top} L^{-1} P,$$

where P is the permutation matrix corresponding to the permutation vector returned by *get_perm()*. If the factorization is in $LDL^\top$ form, the inverse is computed as:

$$A^{-1} = P^\top L^{-\top} D^{-1} L^{-1} P.$$

Added in version 0.2.

logdet

`CholeskyFactor.logdet()`

Compute the (natural) log-determinant of the matrix from its Cholesky factorization.

Returns

logdet (*float*) – The natural logarithm of the determinant of the matrix A that was factorized.

See also

slogdet, *det*, numpy.linalg.slogdet, numpy.linalg.det, scipy.linalg.det

Notes

This function computes the log-determinant of the matrix A from its Cholesky factorization. If the factorization is in LL^T form, the determinant is computed as:

$$\log\det(A) = 2\sum_i \log L_{ii}.$$

If the factorization is in $LDL^\top$ form, the determinant is computed as:

$$\log\det(A) = \sum_i \log D_{ii}.$$

Added in version 0.2.

resymbol

CholeskyFactor.**resymbol**(*A*, *is_permuted=True*)

Recompute the symbolic Cholesky factorization of a sparse matrix.

This function is useful after a series of downdates via *update()* or *rowdel()*, since downdates do not remove any entries in L[1].

Parameters

- **A** (*(N, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric. Only the lower triangular part of A is used, and no check is made for symmetry. The numerical values of A are ignored. Only its non-zero pattern is used.

  Note

  The input matrix A is expected to be the permuted matrix $PAP^\top$, where P is the permutation matrix corresponding to the permutation vector returned by *get_perm()*.

- **is_permuted** (*bool*) – If True (default), the input matrix A is assumed to be permuted by the fill-reducing permutation used in the factorization: $PAP^\top$. If False, A is assumed to be in the original ordering.

Returns

CholeskyFactor – The current object, for method chaining.

See also

cholesky, *ldl*, *update*, *rowadd*, *rowdel*

[1] resymbol.c - CHOLMOD MATLAB resymbolization function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/resymbol.c

References

Added in version 0.5.0.

rowadd

CholeskyFactor.**rowadd**(*k*, *C*)

Add a row to a sparse LDL factorization.

Compute a rank-1 update of a sparse LDL factorization. This method "adds" a row by setting the k^{th} row and column of the original matrix to `C`.

Parameters

- **k** (int $\in [0, N)$) – The row/column index to modify.
- **C** (*(N, 1) csc_array, optional*) – If given, change the factorization such that row and column `k` of the original matrix equal `C`. The number of rows must match that of `L` and `D`.

Returns

- *CholeskyFactor* – The current object, for method chaining.
- *.. versionadded:: 0.5.0*

rowdel

CholeskyFactor.**rowdel**(*k*)

Delete a row from a sparse LDL factorization.

Compute a rank-1 update of a sparse LDL factorization. This method "deletes" a row by setting the k^{th} row and column of the original matrix to the identity.

Parameters

k (int $\in [0, N)$) – The row/column index to modify.

Returns

- *CholeskyFactor* – The current object, for method chaining.
- *.. versionadded:: 0.5.0*

slogdet

CholeskyFactor.**slogdet**()

Compute the sign and (natural) log-determinant of the matrix from its Cholesky factorization.

Returns

- **sign** (*int*) – The sign of the determinant of the matrix *A* that was factorized. This is always 1 for a positive definite matrix.
- **logdet** (*float*) – The natural logarithm of the absolute value of the determinant of the matrix *A* that was factorized.

See also

logdet, *det*, `numpy.linalg.slogdet`, `numpy.linalg.det`, `scipy.linalg.det`

Notes

This function computes the sign and log-determinant of the matrix A from its Cholesky factorization. If the factorization is in LL^T form, the determinant is computed as:

$$\log\det(A) = 2\sum_i \log L_{ii}.$$

If the factorization is in $LDL^\top$ form, the determinant is computed as:

$$\log\det(A) = \sum_i \log D_{ii}.$$

Added in version 0.2.

solve

`CholeskyFactor.solve`(*b*, *system='A'*)

Solve the linear system $Ax = b$ for x, using the factorization.

Parameters

- **b** (*(N,) or (N, K) ndarray or sparse matrix*) – The right-hand side vector or matrix. Must be a type that can be safely cast to the data type of the factor. The number of rows in `b` must be equal to the size of the factor.
- **system** (*str in {'A', 'LDLt', 'LD', 'DLt', 'L', 'Lt', 'D'}, optional*) – The system to solve. Options are:
  - `A`: Solve $Ax = b$.
  - `LDLt`: Solve $LDL^\top x = b$.
  - `LD`: Solve $LDx = b$.
  - `DLt`: Solve $DL^\top x = b$.
  - `L`: Solve $Lx = b$.
  - `Lt`: Solve $L^\top x = b$.
  - `D`: Solve $Dx = b$.

  Here, `L` and `D` are the factors from the LDL factorization of the matrix.

  Note

  Only the `A` system accounts for the permutation used in the factorization. The other systems solve the equations using the factors directly, without applying the permutation.

Returns

x (*(N,) or (N, K) ndarray or sparse matrix*) – The solution vector or matrix, returned in the same format as `b`.

Raises

CholmodNotPositiveDefiniteError – If the matrix A is exactly singular, or singular to working precision.

Notes

This function solves the linear system:

$$R^\top Rx = b,$$

where *R* is the upper triangular factor from the Cholesky factorization of *A*. The input *b* is either dense or sparse, vector or matrix.

If `order` was not `natural` when the factorization was computed, solve the system:

$$P^\top R^\top RPx = b$$

where *P* is the permutation matrix corresponding to the permutation vector. Similarly, if `lower` was True when the factorization was computed, the system solved is:

$$P^\top LL^\top Px = b.$$

If the factorization is in LDL form, the system solved is:

$$P^\top LDL^\top Px = b.$$

This function uses the CHOLMOD library to solve the linear system. It is intended to combine the MATLAB interfaces `cholmod2.m`[1], and `ldlsolve.m`[2].

Added in version 0.5.0.

References

update

`CholeskyFactor.update(`*C*`)`

Multiple-rank update of a sparse LDL factorization.

Compute a update to the factorization of a sparse matrix *A*[1]:

$$L'D'L'^\top = P(A + CC^\top)P^\top$$

where *L* is a lower triangular matrix with unit diagonal, and *D* is a diagonal matrix. The input `C` is a sparse matrix representing the update to the factorization. The fill-reducing permutation is *not* recomputed from the original *A*.

Parameters

C (*(N, K) csc_array*) – The sparse matrix representing the rank-*k* update or downdate to the matrix.

Returns

CholeskyFactor – The current object, for method chaining.

References

Added in version 0.5.0.

[1] `cholmod2.c` - CHOLMOD MATLAB interface https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/cholmod2.c

[2] `ldlsolve.c` - CHOLMOD MATLAB interface https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/ldlsolve.c

[1] `cholmod_updown.c` - CHOLMOD up/downdate function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/Modify/cholmod_updown.c

Symbolic Analysis

symbfact	Symbolic factorization of a sparse matrix for Cholesky or LDL.
etree	Symbolic factorization of a sparse matrix for Cholesky or LDL.

symbfact

sksparse.cholmod.**symbfact**(*A*, *, *kind=None*, *lower=False*, *return_factor=False*)

Symbolic factorization of a sparse matrix for Cholesky or LDL.

This function performs the symbolic factorization of a sparse matrix `A` for either Cholesky or LDL factorization. It computes the elimination tree and analyzes the sparsity pattern of the matrix[1].

Parameters

- **A** (*(N, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric. No check is made for symmetry, so the upper (or lower) triangular part of the matrix is used for the factorization, depending on the `lower` parameter.
- **kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper triangular part of `A` is used, and no check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.
  - `lo`: Lower triangular factorization. Same as `symbfact(A.T)`. Only the lower triangular part of `A` is used, and no check is made for symmetry.

  If `kind` is None, it defaults to `sym`.
- **lower** (*bool, optional*) – If True, the symbolic factorization is performed on the lower triangular part of the matrix. If False, the upper triangular part is used. Default is False (upper triangular).
- **return_factor** (*bool, optional*) – If True, the symbolic factorization returns the structure of the Cholesky factor *L* (or *LD* for LDL factorization) as a sparse matrix. Default is False.

Returns

- **count** (*(N,) ndarray of int*) – The count of nonzeros in each column of the Cholesky factor.
- **h** (*int*) – The height of the elimination tree.
- **parent** (*(N,) ndarray of int*) – The parent of each node in the elimination tree. The root has no parent (parent[0] = -1).
- **post** (*(N,) ndarray of int*) – The postorder of the elimination tree. The first node in the postorder is the root of the tree.
- **L** (*(N, N) csc_array*) – The symbolic factorization of the matrix. Only returned if `return_factor` is True.

[1] `symbfact2.c` - CHOLMOD MATLAB symbolic factorization function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/symbfact2.c

See also

etree

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import cholesky, symbfact
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> L = cholesky(A, lower=True)
>>> count, h, parent, post = symbfact(A)
>>> count
array([3, 3, 4, 3, 3, 4, 4, 3, 3, 2, 1])
>>> np.array_equal(count, np.count_nonzero(L.toarray(), axis=0))
True
>>> h
6
>>> parent
array([ 5,  2,  7,  5,  7,  6,  8,  9,  9, 10, -1])
>>> post
array([ 1,  2,  4,  7,  0,  3,  5,  6,  8,  9, 10])
```

etree

`sksparse.cholmod.`**`etree`**(*A*, ***, *kind=None*, *return_post=False*)

Symbolic factorization of a sparse matrix for Cholesky or LDL.

This function determines the elimination tree of a sparse matrix `A`, and optionally postorders the tree[1].

Parameters

- **A** (*(N, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric. No check is made for symmetry, so the upper (or lower) triangular part of the matrix is used for the factorization, depending on the `lower` parameter.
- **kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper triangular part of `A` is used, and no check is made for symmetry.

[1] `etree2.c` - CHOLMOD MATLAB symbolic factorization function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/etree2.c

- `row`: Unsymmetric factorization of $AA^\top$.
- `col`: Unsymmetric factorization of $A^\top A$.
- `lo`: Lower triangular factorization. Same as `symbfact(A.T)`. Only the lower triangular part of `A` is used, and no check is made for symmetry.

If `kind` is None, it defaults to `sym`.

- **return_post** (*bool, optional*) – If True, the function returns the postorder of the elimination tree. Default is False.

Returns

- **parent** (*(N,) ndarray of int*) – The parent of each node in the elimination tree. The root has no parent (parent[0] = -1).
- **post** (*(N,) ndarray of int, optional*) – The postorder of the elimination tree. The first node in the postorder is the root of the tree.

Added in version 0.5.0.

See also

symbfact

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import etree
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> parent, post = etree(A, return_post=True)
>>> parent
array([ 5,  2,  7,  5,  7,  6,  8,  9,  9, 10, -1])
>>> post
array([ 1,  2,  4,  7,  0,  3,  5,  6,  8,  9, 10])
```

Graph Partitioning

bisect	Compute a node separator for a sparse matrix graph.
metis	Nested dissection ordering of a sparse matrix using METIS.
nesdis	Nested dissection ordering of a sparse matrix.

continues on next page

Table 15 – continued from previous page

SeparatorTree	The separator tree of a sparse matrix graph.

bisect

`sksparse.cholmod.bisect(A, *, kind=None)`

Compute a node separator for a sparse matrix graph.

Parameters

- **A** (*(M, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric if `kind` is None or `"sym"`. No check is made for symmetry.
- **kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper triangular part of `A` is used, and no check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.

  If `kind` is None, it defaults to `sym`.

Returns

s (*(K,) ndarray of int*) – The dimension `K` is either `M` or `N`, depending on the `kind` parameter. The output can take 3 values:

- `0`: The node is in the left subgraph.
- `1`: The node is in the right subgraph.
- `2`: The node is in the separator.

See also

nesdis, *metis*

Notes

This function is based on the SuiteSparse CHOLMOD MATLAB interface[1].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import bisect
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
```

(continues on next page)

[1] `bisect.c` - CHOLMOD MATLAB bisect function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/bisect.c

(continued from previous page)

```
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> s = bisect(A)
>>> s
array([0, 1, 1, 0, 1, 0, 0, 1, 0, 2, 2])
```

metis

sksparse.cholmod.**metis**(*A*, *, *kind=None*)

Nested dissection ordering of a sparse matrix using METIS.

Parameters

- **A** (*(M, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric if `kind` is None or `"sym"`. No check is made for symmetry.
- **kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper triangular part of `A` is used, and no check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.

  If `kind` is None, it defaults to `sym`.

Returns

p (*(M or N,) ndarray of int*) – The permutation vector that gives the nested dissection ordering of the nodes in the graph represented by the sparse matrix `A`.

See also

bisect, *nesdis*

Notes

This function is based on the SuiteSparse CHOLMOD MATLAB interface[1].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
```

(continues on next page)

[1] `metis.c` - CHOLMOD MATLAB metis function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/metis.c

(continued from previous page)

```
>>> from sksparse.cholmod import metis
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> p = metis(A)
>>> p
array([ 8,  3,  6,  0,  5,  2,  4,  7,  1,  9, 10])
```

nesdis

sksparse.cholmod.**nesdis**(*A*, ***, *kind=None*, *return_separator=False*, *nd_small=None*, *nd_components=None*, *nd_oksep=None*, *nd_camd=None*)

Nested dissection ordering of a sparse matrix.

Parameters

- **A** (*(M, N) csc_array*) – The input matrix in Compressed Sparse Column (CSC) format. Must be square and symmetric if `kind` is None or `"sym"`. No check is made for symmetry.
- **kind** (*str in {"sym", "row", "col"}, optional*) – The type of factorization for which to analyze the matrix:
  - `sym`: Symmetric factorization. Only the upper triangular part of `A` is used, and no check is made for symmetry.
  - `row`: Unsymmetric factorization of $AA^\top$.
  - `col`: Unsymmetric factorization of $A^\top A$.

  If `kind` is None, it defaults to `sym`.
- **return_separator** (*bool, optional*) – If True, the function returns the separator tree and component membership vector. Default is False.

Returns

- **p** (*(M or N,) ndarray of int*) – The permutation vector that gives the nested dissection ordering of the nodes in the graph represented by the sparse matrix `A`.
- **septree** (*SeparatorTree, optional*) – The separator tree and component membership vector, returned if `return_separator` is True.

Other Parameters

- **nd_small** (*int, optional*) – The smallest subgraph that should not be partitioned (default is 200).
- **nd_components** (*bool, optional*) – True if connected components should be split independently (default is False).

- **nd_oksep** (*double, optional*) – Controls when a separator is kept. A separator is kept if `nsep < nd_oksep * n`, where `nsep` is the number of nodes in the separator and `n` is the number of nodes in the graph being cut (default is 1).
- **nd_camd** (*int, optional*) – Controls whether the smallest subgraphs should be ordered. If 0, they are not ordered. For the "sym" case, 1 to order by `camd`, 2 to order by `csymamd` (default 1). For other cases: 0 to order naturally, or 1 to order by `colamd`.

See also

bisect, *metis*

Notes

This function is based on the SuiteSparse CHOLMOD MATLAB interface[1].

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import coo_array
>>> from sksparse.cholmod import nesdis
>>> # Create a symmetric positive definite matrix from (Davis, Eqn 2.1)
>>> N = 11
>>> rows = np.array([5, 6, 2, 7, 9, 10, 5, 9, 7, 10, 8, 9, 10, 9, 10, 10])
>>> cols = np.array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 9])
>>> rng = np.random.default_rng(56)
>>> vals = rng.random(len(rows), dtype=np.float64)
>>> L = coo_array((vals, (rows, cols)), shape=(N, N))
>>> A = L + L.T   # make it symmetric
>>> A.setdiag(N)  # make it strongly positive definite
>>> A = A.tocsc()
>>> p, s = nesdis(A, return_separator=True)
>>> p
array([ 1,  4,  6,  8,  0,  3,  5,  2,  9, 10,  7])
>>> s
SeparatorTree(components=1, nodes=11)
```

SeparatorTree

class sksparse.cholmod.**SeparatorTree**(*cp*, *cmember*)

The separator tree of a sparse matrix graph.

This object is typically created by *nesdis()*.

cp

The separator tree, where `C` is the number of components found. The value `cp[c]` is the parent of the component `c` in the separator tree, or `-1` if `c` is the root of the tree. There is a maximum of `N` components, where `N` is the dimension of the input matrix.

[1] `nesdis.c` - CHOLMOD MATLAB nesdis function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/nesdis.c

Type
(C,) numpy.ndarray of int, optional

cmember

The component membership vector, where `cmember[i]` is the component to which node `i` belongs.

Type
(N,) numpy.ndarray of int, optional

Added in version 0.5.0.

Methods

__init__	
prune	Prune the separator tree.

__init__

SeparatorTree.**__init__**(*cp*, *cmember*)

prune

SeparatorTree.**prune**(*, *nd_oksep=None*, *nd_small=None*)

Prune the separator tree.

Parameters

- **nd_oksep** (*double in [0, 1], optional*) – Controls when a separator is kept. A separator is kept if `nsep < nd_oksep * n`, where `nsep` is the number of nodes in the separator and `n` is the number of nodes in the graph being cut (default is 1.0).
- **nd_small** (*int >= 0, optional*) – The smallest subgraph that should not be partitioned (default is 200).

Returns

pruned_septree (*SeparatorTree*) – The pruned separator tree. `cp` will be of length `C'`, where `C' <= C` is the number of components remaining after pruning.

Notes

This function is based on the SuiteSparse CHOLMOD MATLAB interface[1].

References

Exceptions and Warnings

CholmodWarning	Base class for CHOLMOD-related warnings.
CholmodSmallDiagonalWarning	Warning for small diagonal entries.
CholmodError	Base class for CHOLMOD-related errors.
CholmodNotPositiveDefiniteError	Raised when the input matrix is not positive definite.
CholmodNotInstalledError	Raised when the CHOLMOD library is not installed.
CholmodOutOfMemoryError	Raised when CHOLMOD runs out of memory.

continues on next page

[1] `septree.c` - CHOLMOD MATLAB septree function https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CHOLMOD/MATLAB/septree.c

Table 17 – continued from previous page

CholmodOverflowError	Raised when CHOLMOD encounters an integer overflow.
CholmodInvalidInputError	Raised when CHOLMOD receives invalid input.
CholmodGpuProblemError	Raised when CHOLMOD encounters a problem with CUDA.

CholmodWarning

exception sksparse.cholmod.**CholmodWarning**

Bases: Warning

Base class for CHOLMOD-related warnings.

CholmodSmallDiagonalWarning

exception sksparse.cholmod.**CholmodSmallDiagonalWarning**

Bases: *CholmodWarning*

Warning for small diagonal entries.

CholmodError

exception sksparse.cholmod.**CholmodError**

Bases: Exception

Base class for CHOLMOD-related errors.

CholmodNotPositiveDefiniteError

exception sksparse.cholmod.**CholmodNotPositiveDefiniteError**

Bases: *CholmodError*

Raised when the input matrix is not positive definite.

CholmodNotInstalledError

exception sksparse.cholmod.**CholmodNotInstalledError**

Bases: *CholmodError*

Raised when the CHOLMOD library is not installed.

CholmodOutOfMemoryError

exception sksparse.cholmod.**CholmodOutOfMemoryError**

Bases: MemoryError, *CholmodError*

Raised when CHOLMOD runs out of memory.

CholmodOverflowError

exception sksparse.cholmod.**CholmodOverflowError**

Bases: *CholmodError*

Raised when CHOLMOD encounters an integer overflow.

CholmodInvalidInputError

exception sksparse.cholmod.**CholmodInvalidInputError**

Bases: *CholmodError*

Raised when CHOLMOD receives invalid input.

CholmodGpuProblemError

exception sksparse.cholmod.**CholmodGpuProblemError**

Bases: *CholmodError*

Raised when CHOLMOD encounters a problem with CUDA.

References

- SuiteSparse homepage
- SuiteSparse CHOLMOD

4.3.6 Column Approximate Minimum Degree (COLAMD) Ordering (*sksparse.colamd*)

Added in version 0.5.0.

Python interface to the Column Approximate Minimum Degree (COLAMD) ordering algorithm.

Interface

colamd(A[, dense_row_thresh, ...])	Compute the column approximate minimum degree ordering of a sparse matrix.
symamd(A[, dense_row_thresh, ...])	Compute the column approximate minimum degree ordering of a sparse symmetric matrix.
colamd_get_defaults()	Get the default knobs for COLAMD.

colamd

sksparse.colamd.**colamd**(*A*, *dense_row_thresh=None*, *dense_col_thresh=None*, *aggressive=None*, *return_info=False*)

Compute the column approximate minimum degree ordering of a sparse matrix.

Adapted from the COLAMD documentation[1]:

> This function computes a column ordering for a sparse matrix *A* that is appropriate for LU factorization of symmetric or unsymmetric matrices, QR factorization, least squares, interior point methods for linear programming problems, and other related problems.
>
> COLAMD computes a permutation Q such that the Cholesky factorization of $(AQ)^\top(AQ)$ has less fill-in and requires fewer floating point operations than $A^\top A$. This also provides a good ordering for sparse partial pivoting methods, $P(AQ) = LU$, where Q is computed prior to numerical factorization, and P is computed during numerical factorization via conventional partial pivoting with row interchanges.
>
> **Parameters**

[1] `colamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/COLAMD/Source/colamd.c

- **A** (*(M, N) array_like or sparse matrix*) – The input matrix for which to compute the column ordering. Must be 2D and convertible to CSC format. Need not be square.
- **dense_row_thresh, dense_col_thresh** (*float, optional*) – Threshold for considering a row/column dense. If None, use the default value from COLAMD. The default value is 10. The actual number of entries in a row/column is to be considered "dense" is `max(dense_row_thresh * sqrt(M), 16)` where `M` is the number of rows (or `N` for columns). Dense rows/columns are ignored during ordering and moved to the end of the matrix.
- **aggressive** (*bool, optional*) – If True, use aggressive absorption. If None, uses the default value from COLAMD. The default value is True.

Returns

- **q** ((N,) `ndarray`) – The permutation vector.
- **stats** (*COLAMDStats*, optional) – If `return_info` is True, returns an object containing statistics about the ordering.

See also

symamd, *ccolamd*, *csymamd*

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array
>>> from sksparse.colamd import colamd
>>> # Create a non-symmetric matrix
>>> N = 11
>>> rng = np.random.default_rng(56)
>>> A = random_array((N, N - 3), density=0.5, format='csc', rng=rng)
>>> A.setdiag(N)  # make the diagonal non-zero
>>> p, info = colamd(A, return_info=True)
>>> p
array([0, 3, 5, 6, 7, 1, 2, 4], dtype=int32)
>>> info
COLAMDStats(N_rows_ignored=0, N_cols_ignored=0, Ncmpa=0, status=0, info1=-1,
    info2=-1, info3=0)
```

symamd

sksparse.colamd.**symamd**(*A*, *dense_row_thresh=None*, *dense_col_thresh=None*, *aggressive=None*, *return_info=False*)

Compute the column approximate minimum degree ordering of a sparse symmetric matrix.

Adapted from the COLAMD documentation[1]:

[1] `colamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/COLAMD/Source/colamd.c

This function computes an approximate minimum degree ordering for Cholesky factorization of symmetric matrices.

Symamd computes a permutation P of a symmetric matrix A such that the Cholesky factorization of $PAP^\top$ has less fill-in and requires fewer floating point operations than A. Symamd constructs a matrix M such that $M^\top M$ has the same nonzero pattern of A, and then orders the columns of M using colamd. The column ordering of M is then returned as the row and column ordering P of A.

Parameters

A (*(N, N) {array_like, sparse matrix}*) – The input matrix for which to compute the column ordering. Must be 2D, square, and convertible to CSC format.

> **Note**
>
> This routine only accesses the lower triangular part of `A`, which is *assumed* to be symmetric. If it is not, the results may be incorrect or undefined.

dense_row_thresh, dense_col_thresh

[float, optional] Threshold for considering a row/column dense. If None, use the default value from COLAMD. The default value is 10. The actual number of entries in a row/column is to be considered "dense" is `max(dense_row_thresh * sqrt(M), 16)` where `M` is the number of rows (or `N` for columns). Dense rows/columns are ignored during ordering and moved to the end of the matrix.

aggressive

[bool, optional] If True, use aggressive absorption. If None, uses the default value from COLAMD. The default value is True.

Returns

- **q** ((N,) ndarray) – The permutation vector.
- **stats** (*COLAMDStats*, optional) – If `return_info` is True, returns an object containing statistics about the ordering.

> **See also**
>
> *colamd*, *ccolamd*, *csymamd*

Added in version 0.5.0.

References

Examples

```
>>> import numpy as np
>>> from scipy.sparse import random_array
>>> from sksparse.colamd import symamd
>>> # Create a non-symmetric matrix
>>> N = 11
>>> rng = np.random.default_rng(56)
>>> A = random_array((N, N - 3), density=0.5, format='csc', rng=rng)
>>> A.setdiag(N)           # make the diagonal non-zero
>>> A = (A.T @ A).tocsc()  # make it symmetric
```

(continues on next page)

(continued from previous page)

```
>>> p, info = symamd(A, return_info=True)
>>> p
array([4, 6, 7, 0, 1, 2, 3, 5], dtype=int32)
>>> info
COLAMDStats(N_rows_ignored=0, N_cols_ignored=0, Ncmpa=0, status=0, info1=-1,
    info2=-1, info3=0)
```

colamd_get_defaults

sksparse.colamd.**colamd_get_defaults**()

Get the default knobs for COLAMD.

Returns

knobs (*dict*) – A dictionary containing the default knobs for COLAMD.

The keys are:

- 'dense_row_thresh': Threshold for considering a row/column dense. Rows with more than `max(dense_row_thresh * sqrt(M), 16)` entries are permuted to the end of the matrix.
- 'dense_col_thresh': Like *dense_row_thresh*, but for columns.
- 'aggressive': Default value for the aggressive knob.

Added in version 0.5.0.

Exceptions and Warnings

COLAMDError	Base class for COLAMD errors.
COLAMDValueError	Raised when COLAMD encounters a value error.
COLAMDMemoryError	Raised when COLAMD runs out of memory.
COLAMDInternalError	Raised when COLAMD encounters an internal error.
COLAMDStats(N_rows_ignored, N_cols_ignored, ...)	Information statistics returned by the COLAMD algorithm.

COLAMDError

exception sksparse.colamd.**COLAMDError**

Bases: Exception

Base class for COLAMD errors.

COLAMDValueError

exception sksparse.colamd.**COLAMDValueError**

Bases: *COLAMDError*, ValueError

Raised when COLAMD encounters a value error.

COLAMDMemoryError

exception sksparse.colamd.**COLAMDMemoryError**

Bases: *COLAMDError*, MemoryError

Raised when COLAMD runs out of memory.

COLAMDInternalError

exception sksparse.colamd.**COLAMDInternalError**

Bases: *COLAMDError*, RuntimeError

Raised when COLAMD encounters an internal error.

COLAMDStats

class sksparse.colamd.**COLAMDStats**(*N_rows_ignored: int*, *N_cols_ignored: int*, *Ncmpa: int*, *status: int*, *info1: int*, *info2: int*, *info3: int*)

Information statistics returned by the COLAMD algorithm.

This class wraps the contents of the `stats` array returned by C `colamd()` into a Python dataclass.

N_rows_ignored

The number of dense or empty rows ignored in the ordering.

Type
int

N_cols_ignored

The number of dense or empty columns ignored in the ordering.

Type
int

Ncmpa

The number of garbage collections performed.

Type
int

status

Status code indicating the result of the COLAMD operation. If non-zero, `colamd` will throw an appropriate exception that interprets this status code.

Type
int

The following fields take on different meanings depending on the value of

``status``

info1

Value of `status`:

- 0: the highest numbered column that is unsorted or has duplicate entries.
- -3: the value of `n_row`.
- -4: the value of `n_col`.
- -5: the value of `nnz == p[n_col]`.
- -6: the value of `p[0]`.

- -7: the required `Alen` value.
- -8: the column with negative entries.
- -9: the column with a row index out of bounds.

Type
int

info2

Value of `status`:

- 0: the last seen duplicate or unsorted row index.
- -7: the actual `Alen` value.
- -9: the bad row index.

Type
int

info3

Value of `status`:

- 0: the number of duplicates or unsorted row indices.
- -9: `n_row`.

Type
int

Notes

Field descriptions are adapted from SuiteSparse `colamd.c`[1].

Added in version 0.5.0.

References

Methods

__init__	
from_array	Create a COLAMDStats instance from an array.

__init__

COLAMDStats.**__init__**(*N_rows_ignored: int*, *N_cols_ignored: int*, *Ncmpa: int*, *status: int*, *info1: int*, *info2: int*, *info3: int*) → None

from_array

classmethod COLAMDStats.**from_array**(*stats: ndarray*) → *COLAMDStats*

Create a COLAMDStats instance from an array.

[1] `colamd.c` - SuiteSparse AMD source file. https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/COLAMD/Source/colamd.c

References

- SuiteSparse homepage
- SuiteSparse COLAMD
- COLAMD Algorithm Publications:
  - T. A. Davis, J. R. Gilbert, S. Larimore, E. Ng, An approximate column minimum degree ordering algorithm, *ACM Transactions on Mathematical Software*, vol. 30, no. 3., pp. 353-376, 2004.
  - T. A. Davis, J. R. Gilbert, S. Larimore, E. Ng, Algorithm 836: COLAMD, an approximate column minimum degree ordering algorithm, *ACM Transactions on Mathematical Software*, vol. 30, no. 3., pp. 377-380, 2004.

4.3.7 Clark Kent LU Decomposition (`sksparse.klu`)

Added in version 0.5.0.

An interface to the SuiteSparse KLU package, which computes the LU factorization and solves systems of equations for sparse, possibly non-symmetric, indefinite matrices.

Function Interface

`klu_solve`	Solve a linear system using KLU.

klu_solve

sksparse.klu.**klu_solve**(*A*, *b*, *, *control=None*, *transpose=False*, *rhs_batch_size=100*, ***kwargs*)

Solve a linear system using KLU.

This function solves a linear system for x given the right-hand side b as either a vector or a matrix with multiple right-hand sides.

If `transpose=False`, solve

$$Ax = b$$

or, if `transpose=True`, solve

$$xA = b \iff A^\top x^\top = b^\top.$$

This is a convenience function that creates a `KLUFactor` object, computes the numeric factorization, and solves the linear system with `KLUFactor.solve()`.

Parameters

- **A** (*(N, N) numpy.ndarray or sparse array*) – The input matrix to factorize.
- **b** (*(N,) or (N, K) numpy.ndarray*) – The right-hand side vector or matrix.
- **control** (`KLUControl`, optional) – An optional `KLUControl` object to set the factorization parameters. If not provided, default parameters are used.
- **transpose** (*bool, optional*) – If True, solve $xA = b$, otherwise, solve $Ax = b$.
- **rhs_batch_size** (*int, optional*) – If b is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.

- ****kwargs** – Additional keyword arguments passed to the *KLUControl* constructor.

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix of the same type and shape as the input right-hand side b.

See also

KLUFactor, *klu_solve*

Added in version 0.5.0.

Examples

See: Davis, Timothy A. (2006). Direct Methods for Sparse Linear Systems, p 74 (Figure 5.1)

```
>>> import numpy as np
>>> from scipy import sparse
>>> from sksparse.klu import klu_solve
>>> N = 8
>>> rows = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 3, 6, 1, 6, 0, 2, 5, 7, 4, 7, 0, 1, 3, 7, 5, 6],
...    dtype=np.int32,
...)
>>> cols = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7],
...    dtype=np.int32,
...)
>>> vals = np.ones(len(rows), dtype=np.float64)
>>> vals[:7] = np.arange(1, 8, dtype=np.float64)  # make diagonal entries non-unit
>>> A = sparse.csc_array((vals, (rows, cols)), shape=(N, N))
>>> A
<Compressed Sparse Column sparse array of dtype 'float64'
        with 23 stored elements and shape (8, 8)>
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = klu_solve(A, b)
>>> np.allclose(x, expect_x)
True
```

Object Interface

klu_factor	Compute the LU factorization of a sparse matrix using KLU.
KLUFactor	Class to compute and store the KLU factorization of a sparse matrix.
KLUInfo	A dataclass to store KLU information.
KLUControl	A dataclass to set KLU control parameters.

klu_factor

sksparse.klu.**klu_factor**(*A*, *, *control=None*, ***kwargs*)

Compute the LU factorization of a sparse matrix using KLU.

This is a convenience function that creates a *KLUFactor* object, computes the numeric factorization, and returns the resulting object.

Parameters

- **A** (*(M, N) numpy.ndarray or sparse array*) – The input matrix to factorize.
- **control** (*KLUControl*, optional) – An optional *KLUControl* object to set the factorization parameters. If not provided, default parameters are used.
- ****kwargs** – Additional keyword arguments passed to the *KLUControl* constructor.

Returns

KLUFactor – The LU factorization of the input matrix.

Raises

KLUSingularMatrixWarning – If the matrix is exactly singular.

See also

KLUFactor, *klu_solve*

Added in version 0.5.0.

Examples

See: Davis, Timothy A. (2006). Direct Methods for Sparse Linear Systems, p 74 (Figure 5.1)

```
>>> import numpy as np
>>> from scipy import sparse
>>> from sksparse.klu import klu_factor
>>> N = 8
>>> rows = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 3, 6, 1, 6, 0, 2, 5, 7, 4, 7, 0, 1, 3, 7, 5, 6],
...    dtype=np.int32,
...)
>>> cols = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7],
...    dtype=np.int32,
...)
>>> vals = np.ones(len(rows), dtype=np.float64)
>>> vals[:7] = np.arange(1, 8, dtype=np.float64)  # make diagonal entries non-unit
>>> A = sparse.csc_array((vals, (rows, cols)), shape=(N, N))
>>> A
<Compressed Sparse Column sparse array of dtype 'float64'
        with 23 stored elements and shape (8, 8)>
>>> # Compute the LU factorization
>>> f = klu_factor(A)
>>> f
```

(continues on next page)

(continued from previous page)

```
<KLUFactor numeric factor of dtype 'float64' with 'int32' indices:
    L: (8, 8) with 16 stored elements
    U: (8, 8) with 17 stored elements>
>>> L, U, p, q, r, F, _ = f  # unpack the factorization
>>> LUF = (L @ U + F).toarray()
>>> RPAQ = (r[:, np.newaxis] * A[p[:, np.newaxis], q]).toarray()
>>> np.allclose(LUF, RPAQ)
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

KLUFactor

class sksparse.klu.**KLUFactor**

Class to compute and store the KLU factorization of a sparse matrix.

The constructor computes the symbolic analysis of a sparse matrix A and determines a fill-reducing ordering such that:

$$LU + F = RPAQ.$$

The numeric factorization is not computed until *factorize()* is called.

Note

Note that the use of the scale factor R differs between KLU and UMFPACK:

$$LU = PR_{\text{umf}}AQ$$
(UMFPACK),
$$LU + F = R_{\text{klu}}PAQ$$
(KLU).

They are related by $R_{\text{klu}} = PR_{\text{umf}}P^{\top}$.

is_numeric

Whether the numeric factorization has been computed.

Type
bool

lnz

The number of nonzeros in the L factor.

Type
int

unz

The number of nonzeros in the U factor.

Type
int

`nzoff`

The number of nonzeros in the F factor.

Type
int

`nblocks`

The number of blocks in the BTF ordering of the matrix.

Type
int

`nnz`

The number of nonzeros in the original matrix.

Type
int

`shape`

The shape of the original matrix.

Type
tuple of int

`dtype`

The data type of the matrix entries (`float64` or `complex128`).

Type
numpy.dtype

`itype`

The integer type used for indexing (`int32` or `int64`).

Type
numpy.dtype

`L`

The L factor as a sparse CSC matrix.

Type
scipy.sparse.csc_array

`U`

The U factor as a sparse CSC matrix.

Type
scipy.sparse.csc_array

`F`

The F factor as a sparse CSC matrix.

Type
scipy.sparse.csc_array

`perm_r, perm_c`

The row and column permutation arrays.

Type
numpy.ndarray

rscale

The row scaling array.

Type
numpy.ndarray

rblocks

The row blocks in the BTF ordering.

Type
numpy.ndarray of int

info

An object containing information about the factorization.

Type
KLUInfo

Notes

This object is an interface to the SuiteSparse KLU library[1].

Added in version 0.5.0.

References

Methods

__init__	Compute the KLU factorization of a sparse matrix.
copy	Return a deep copy of the current KLUFactor object.
factorize	Compute the numeric factorization of the matrix.
solve	Solve a linear system using the KLU factorization.

__init__

KLUFactor.**__init__**()

Compute the KLU factorization of a sparse matrix.

Parameters

- **A** (*(N, N) numpy.ndarray or sparse array*) – The input matrix. Any object that can be converted to a `csc_array` is accepted.
- **control** (*KLUControl*, optional) – An optional *KLUControl* object to set the factorization parameters. If not provided, default parameters are used.

copy

KLUFactor.**copy**()

Return a deep copy of the current KLUFactor object.

[1] SuiteSparse KLU https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/KLU

factorize

KLUFactor.**factorize**(*A*)

Compute the numeric factorization of the matrix.

Computes the numeric factorization of a sparse matrix A and determines a fill-reducing ordering such that:

$$LU + F = RPAQ.$$

If given, the matrix A must have the same shape and nonzero pattern as the one used to create this *KLUFactor* object, but need not have the same values.

> **Warning**
>
> No check is made on the non-zero structure of the input matrix, so if it is different from the one used for the symbolic analysis, the results will be incorrect without raising an error.

Parameters

A (*(N, N) numpy.ndarray or sparse array*) – The input matrix. Must have the same shape and nonzero pattern as the matrix used to create this *KLUFactor* object. If not provided, the original matrix given to the constructor will be used.

Returns

KLUFactor – The current object, for method chaining.

solve

KLUFactor.**solve**(*b*, ***, *transpose=False*, *rhs_batch_size=100*)

Solve a linear system using the KLU factorization.

This method solves a linear system for x given the right-hand side b as either a vector or a matrix with multiple right-hand sides.

If `transpose=False`, solve

$$Ax = b$$

or, if `transpose=True`, solve

$$xA = b \iff A^\top x^\top = b^\top.$$

The method uses the LU factorization of A previously computed by *factorize()*.

Parameters

- **b** (*(N,) or (N, K) numpy.ndarray*) – The right-hand side vector or matrix.
- **transpose** (*bool, optional*) – If True, solve $xA = b$, otherwise, solve $Ax = b$.
- **rhs_batch_size** (*int, optional*) – If `b` is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix. If `b` is a 1D array, then `x` is returned as a 1D array. If `b` is a 2D array with `K` columns, then `x` is returned as a 2D array with `K` columns. If `b` is a sparse array, then `x` is also returned as a sparse array.

KLUInfo

class sksparse.klu.**KLUInfo**

A dataclass to store KLU information.

noffdiag

Number of off-diagonal entries in the matrix.

Type
int

nrealloc

Number of memory reallocations during factorization.

Type
int

rcond

Estimate of the reciprocal of the condition number.

Type
float

singular_col

Index of the first singular column, if any.

Type
int

rgrowth

The reciprocal pivot growth factor.

Type
float

flops

Estimated number of floating-point operations.

Type
int

nblocks

Number of blocks in the BTF ordering of the matrix.

Type
int

ordering

The fill-reducing ordering used.

Type
str

scale

The row-scaling method used.

Type
str

lnz

Number of nonzeros in the L factor.

Type
int

unz

Number of nonzeros in the U factor.

Type
int

nzoff

Number of nonzeros in the F factor ("offset").

Type
int

tol

The pivot tolerance used.

Type
float

mempeak

Peak memory usage in bytes.

Type
int

Added in version 0.5.0.

Methods

__init__

__init__

KLUInfo.**__init__**(**args*, ***kwargs*)

KLUControl

class sksparse.klu.**KLUControl**

A dataclass to set KLU control parameters.

tol

The pivot tolerance. Default is `None`, which uses the KLU default of `0.001`.

Type
float

memgrow

The memory growth factor. Default is `None`, which uses the KLU default of `1.2`.

Type
float

initmem_amd

The initial memory allocation factor for AMD. Default is `None`, which uses the `KLU` default of `1.2`.

Type
float

initmem

The initial memory allocation factor for the numeric factorization. Default is `None`, which uses the `KLU` default of `10`.

Type
float

maxwork

The maximum work done by BTF. Default is `None`, which uses the `KLU` default of `0`, or unlimited.

Type
float

btf

Whether to use BTF pre-ordering. Default is `None`, which uses the `KLU` default of `True`.

Type
bool

ordering

The fill-reducing ordering. Accepted values are:

- `AMD`: Approximate Minimum Degree ordering.
- `COLAMD` : Column Approximate Minimum Degree ordering.
- `user_perm`: User-provided ordering (not yet supported).
- `user_func`: User-defined ordering function (not yet supported).

Default is `None`, which uses the `KLU` default setting of `AMD`.

Type
str

scale

The row-scaling method. Accepted values are:

- `none_no_check`
- `none`
- `sum`
- `max`

Default is `None`, which uses the `KLU` default setting of `max`.

Type
str

Added in version 0.5.0.

Methods

__init__

__init__

KLUControl.**__init__**()

Warnings and Exceptions

KLUWarning	Base warning for KLU-related warnings.
KLUSingularMatrixWarning	Warning raised when a singular matrix is encountered.
KLUError	Base exception for KLU-related errors.
KLUOutOfMemoryError	Exception raised when KLU runs out of memory.
KLUInvalidError	Exception raised for invalid inputs to KLU.
KLUOverflowError	Exception raised when KLU encounters an overflow.

KLUWarning

exception sksparse.klu.**KLUWarning**

Bases: Warning

Base warning for KLU-related warnings.

KLUSingularMatrixWarning

exception sksparse.klu.**KLUSingularMatrixWarning**

Bases: *KLUWarning*

Warning raised when a singular matrix is encountered.

KLUError

exception sksparse.klu.**KLUError**

Bases: Exception

Base exception for KLU-related errors.

KLUOutOfMemoryError

exception sksparse.klu.**KLUOutOfMemoryError**

Bases: MemoryError, *KLUError*

Exception raised when KLU runs out of memory.

KLUInvalidError

exception sksparse.klu.**KLUInvalidError**

Bases: *KLUError*

Exception raised for invalid inputs to KLU.

KLUOverflowError

exception sksparse.klu.**KLUOverflowError**

Bases: `OverflowError`, *KLUError*

Exception raised when KLU encounters an overflow.

References

- SuiteSparse homepage
- SuiteSparse KLU

4.3.8 Sparse QR Decomposition (*sksparse.spqr*)

Added in version 0.5.0.

An interface to the SuiteSparse SPQR package, which computes the QR factorization and solves systems of equations for sparse, possibly non-square, non-symmetric, indefinite matrices.

Function Interface

spqr	Compute the QR factorization.
spqr_qmult	Multiply by Q using the Householder representation.
spqr_solve	Solve a linear system using the SPQR factorization.
SPQRHouseholder	A class to hold the Householder representation of Q.

spqr

sksparse.spqr.**spqr**(*A*, ***, *mode='full'*, *order=None*, *tol=None*)

Compute the QR factorization.

This function computes the QR factorization of a sparse matrix A such that

$$QR = AE$$

where Q is an orthogonal matrix and R is an upper-triangular matrix. E is a column permutation matrix that reduces fill-in during the factorization.

Parameters

- **A** (*(M, N) array_like or sparse array*) – An array convertible to a sparse matrix.
- **mode** (*{'full', 'r', 'economic', 'householder'}, optional*) – The mode of the returned Q and R matrices. Options are:
  - `full`: `Q` is size `(M, M)`, `R` is size `(M, N)`.
  - `economic`: `Q` is size `(M, K)`, `R` is size `(K, N)`, where `K = min(M, N)`.
  - `r`: Only return the upper-triangular matrix `R`.
  - `householder`: Return the Householder vectors and coefficients used to build `Q`. This option is similar to `mode='raw'` in `scipy.linalg.qr()`.
- **order** (*str, optional*) – The column ordering strategy to use. Let S be the matrix A with singleton rows/columns removed, the ordering options are:
  - `default`: COLAMD(S),

  - `fixed`: identity permutation (*i.e.* no singletons removed),
  - `natural`: singletons removed, but no fill-reducing ordering applied,
  - `colamd`: COLAMD(S),
  - `amd`: AMD($S^\top S$),
  - `metis`: METIS($S^\top S$),
  - `best`: try all of `amd`, `colamd`, `metis` and pick the best,
  - `cholmod`: Same as `best`,
  - `bestamd`: try `amd` and `colamd` and pick the best.
- **tol** (*float, optional*) – If the 2-norm of a column in `A` is less than `tol`, that column is considered to be a zero column. If `tol = 0`, no columns are treated as zero. If `None`, the default tolerance is used. The default is `tol` = $20\epsilon(M+N)\sqrt{\max \mathrm{diag}(A^\top A)}$, where ϵ is the machine precision.

Returns

- **Q** (*csc_array*) – The orthogonal matrix Q. Shape (M, M) or (M, K) if `mode='economic'`. Not returned if `mode='r'`. Replaced by *SPQRHouseholder* if `mode='householder'`.
- **R** (*csc_array*) – The upper-triangular matrix R. Shape (M, N) or (K, N) if `mode in ['economic', 'householder']`, where K = min(M, N).
- **P** (*ndarray of int*) – The permutation vector of shape (N,).

See also

SPQRFactor, *spqr_factor*, *spqr_qmult*, *spqr_solve*

Notes

This function is part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

spqr_qmult

sksparse.spqr.**spqr_qmult**(*house*, *X*, *method='QX'*)

Multiply by Q using the Householder representation.

Parameters

- **house** (*SPQRHouseholder or tuple*) – A tuple `(H, tau, v)` representing the Householder vectors `H`, coefficients `tau`, and the column permutation vector `v`. Typically, these are created from `Ht, R, p = spqr(A, mode='householder')`.
- **X** (*(M, N) numpy.ndarray or sparse array*) – The matrix to be multiplied. Must have compatible shape with `Q`.
- **method** (*str , optional*) – The multiplication method. Options are:
  - `QX` : compute QX

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

- `QTX` : compute $Q^\top X$
- `XQ` : compute XQ
- `XQT` : compute $XQ^\top$

Default is `QX`. The transpose is the conjugate transpose for complex data.

Returns

Y (*(M, N) numpy.ndarray or sparse array*) – The result of the multiplication. If `X` is a sparse array, then `Y` is also returned as a sparse array.

See also

SPQRFactor, *spqr_factor*, *spqr*, *spqr_solve*

Notes

This function is part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

spqr_solve

`sksparse.spqr.`**`spqr_solve`**(*A*, *b*, **, transpose=False, min2norm=True, rhs_batch_size=100*)

Solve a linear system using the SPQR factorization.

Solve a linear system for x given the right-hand side b as either a vector or a matrix with multiple right-hand sides.

If `transpose=False`, solve

$$Ax = b$$

or, if `transpose=True`, solve

$$A^\top x = b.$$

Parameters

- **A** (*(M, N) array_like or sparse array*) – An array convertible to a sparse matrix.
- **b** (*(M,) or (M, K) numpy.ndarray*) – The right-hand side vector or matrix. `M` should be the number of rows in `A` if `transpose=False`, otherwise the number of columns.
- **transpose** (*bool, optional*) – Whether to solve the transposed system. Default is False.
- **min2norm** (*bool, optional*) – If True, compute the minimum 2-norm solution when `A` is underdetermined. `transpose` is ignored in this case. Default is True. If False, the solution of an underdetermined system is not guaranteed to be the minimum 2-norm solution.
- **rhs_batch_size** (*int, optional*) – If `b` is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix. If `b` is a 1D array, then `x` is returned as a 1D array. If `b` is a 2D array with `K` columns, then `x` is returned as a 2D array with `K` columns. If `b` is a sparse array, then `x` is also returned as a sparse array. `N` is the number of columns in `A` if `transpose=False`, otherwise the number of rows.

See also

SPQRFactor, *spqr_factor*, *spqr*, *spqr_qmult*

Notes

Part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

SPQRHouseholder

class sksparse.spqr.**SPQRHouseholder**(*H: csc_array*, *tau: ndarray*, *perm: ndarray*)

A class to hold the Householder representation of Q.

H

The Householder vectors stored in a sparse matrix.

Type
csc_array

tau

The Householder coefficients.

Type
ndarray of float

perm

The column permutation vector.

Type
ndarray of int

Added in version 0.5.0.

Methods

__init__	
count	Return number of occurrences of value.
index	Return first index of value.

__init__

SPQRHouseholder.**__init__**()

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

count

SPQRHouseholder.**count**(*value*, */ (Positional-only parameter separator (PEP 570))*)

Return number of occurrences of value.

index

SPQRHouseholder.**index**(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

Object Interface

spqr_factor	Compute the SPQR factorization of a sparse matrix.
SPQRFactor	The main object used for creating and manipulating SPQR factorizations.
SPQRInfo	A dataclass to hold SPQR info statistics.

spqr_factor

sksparse.spqr.**spqr_factor**(*A*, *, *use_singletons=False*, *order=None*, *tol=None*)

Compute the SPQR factorization of a sparse matrix.

Compute the numeric factorization of the matrix and determine a fill-reducing ordering such that:

$$QR = AE$$

where E is a column permutation matrix, Q is an orthogonal matrix, and R is an upper-triangular matrix.

This function returns a *SPQRFactor* object that contains the SPQR factorization of the input matrix. It is not currently possible to extract the individual factors Q and R explicitly, but the object provides methods to reuse the factorization to solve linear systems or multiply by Q.

Parameters

- **A** (*(M, N) array_like or sparse array*) – An array convertible to a sparse matrix.
- **use_singletons** (*bool, optional*) – If True, directly compute the numeric factorization to exploit singleton rows. Otherwise, only perform symbolic analysis. Default is False, so that the factor can be reused efficiently for multiple numeric factorizations.
- **order** (*str, optional*) – The column ordering strategy to use. Let S be the matrix A with singleton rows/columns removed, the ordering options are:
  - `default`: COLAMD(S),
  - `fixed`: identity permutation (*i.e.* no singletons removed),
  - `natural`: singletons removed, but no fill-reducing ordering applied,
  - `colamd`: COLAMD(S),
  - `amd`: AMD($S^\top S$),
  - `metis`: METIS($S^\top S$),
  - `best`: try all of `amd`, `colamd`, `metis` and pick the best,
  - `cholmod`: Same as `best`,

- `bestamd`: try `amd` and `colamd` and pick the best.

- **tol** (*float, optional*) – If the 2-norm of a column in `A` is less than `tol`, that column is considered to be a zero column. If `tol = 0`, no columns are treated as zero. If `None`, the default tolerance is used. The default is `tol` = $20\epsilon(M + N)\sqrt{\max \operatorname{diag}(A^\top A)}$, where ϵ is the machine precision.

Returns

SPQRFactor – The SPQR factorization of the input matrix.

See also

SPQRFactor, *spqr*, *spqr_qmult*, *spqr_solve*

Notes

This function is part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

SPQRFactor

class sksparse.spqr.**SPQRFactor**

The main object used for creating and manipulating SPQR factorizations.

The constructor computes the sybolic analysis of the matrix and determines a fill-reducing ordering such that:

$$QR = AE$$

where E is a column permutation matrix, Q is an orthogonal matrix, and R is an upper-triangular matrix.

The numerical factorization is computed in one of two ways:

1. **by setting `use_singletons=True` in the constructor, which computes** both the symbolic and numeric factorizations at once, or
2. **by calling *`SPQRFactor.factorize()`*, which computes the numeric** factorization after symbolic analysis has been performed.

The first method is useful when factoring a single matrix, but solving multiple right-hand sides.

The second method is useful when factoring multiple matrices with the same sparsity pattern but different numerical values.

Parameters

- **A** (*(M, N) array_like or sparse array*) – An array convertible to a sparse matrix.
- **use_singletons** (*bool, optional*) – If True, directly compute the numeric factorization to exploit singleton rows. Otherwise, only perform symbolic analysis. Default is False.
- **order** (*str, optional*) – The column ordering strategy to use. Let S be the matrix A with singleton rows/columns removed, the ordering options are:
  - `default`: COLAMD(S),
  - `fixed`: identity permutation (*i.e.* no singletons removed),

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

  - `natural`: singletons removed, but no fill-reducing ordering applied,
  - `colamd`: COLAMD(S),
  - `amd`: AMD($S^\top S$),
  - `metis`: METIS($S^\top S$),
  - `best`: try all of `amd`, `colamd`, `metis` and pick the best,
  - `cholmod`: Same as `best`,
  - `bestamd`: try `amd` and `colamd` and pick the best.
- **tol** (*float, optional*) – If the 2-norm of a column in `A` is less than `tol`, that column is considered to be a zero column. If `tol = 0`, no columns are treated as zero. If `None`, the default tolerance is used. The default is `tol` = $20\epsilon(M + N)\sqrt{\max \operatorname{diag}(A^\top A)}$, where ϵ is the machine precision.

is_numeric

Whether the numeric factorization has been computed.

Type
bool

shape

The shape of the input matrix (M, N).

Type
tuple

itype

The integer type used for indices (`int32` or `int64`).

Type
dtype

dtype

The data type of the matrix (`float64` or `complex128`).

Type
dtype

rank

The rank of the matrix as determined by SPQR.

Type
int

perm

The combined singleton and fill-reducing column permutation vector.

Type
ndarray of int

info

An object containing various SPQR statistics.

Type
SPQRInfo

See also

spqr_factor, *spqr*, *spqr_qmult*, *spqr_solve*

Notes

This object is an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

Methods

__init__	Initialize the SPQRFactor object and perform symbolic analysis.
copy	Return a deep copy of the SPQRFactor object.
factorize	Compute the numeric factorization of the matrix.
qmult	Multiply by Q using the Householder representation.
solve	Solve a linear system using the SPQR factorization.

__init__

SPQRFactor.**__init__**()

Initialize the SPQRFactor object and perform symbolic analysis.

copy

SPQRFactor.**copy**()

Return a deep copy of the SPQRFactor object.

factorize

SPQRFactor.**factorize**(*A*, ***, *tol=None*)

Compute the numeric factorization of the matrix.

Parameters

- **A** (*(M, N) array_like or sparse array, optional*) – An array convertible to a sparse matrix. If None, the numeric factorization is computed for the matrix used in the constructor. If `A` is provided, it must have the same sparsity pattern as the matrix used in the constructor.
- **tol** (*float, optional*) – If the 2-norm of a column in `A` is less than `tol`, that column is considered to be a zero column. If `tol = 0`, no columns are treated as zero. If `None`, the default tolerance is used. The default is `tol` = $20\epsilon(M+N)\sqrt{\max \operatorname{diag}(A^\top A)}$, where ϵ is the machine precision.

Returns

SPQRFactor – The current object with the numeric factorization computed.

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

qmult

SPQRFactor.**qmult**(*X*, *method='QX'*)

Multiply by Q using the Householder representation.

Parameters

- **X** (*(M, N) numpy.ndarray or sparse array*) – The matrix to be multiplied. Must have compatible shape with `Q`.
- **method** (*str , optional*) – The multiplication method. Options are:
  - `QX` : compute QX
  - `QTX` : compute $Q^\top X$
  - `XQ` : compute XQ
  - `XQT` : compute $XQ^\top$

  Default is `QX`. The transpose is the conjugate transpose for complex data.

Returns

Y (*(M, N) numpy.ndarray or sparse array*) – The result of the multiplication. If `X` is a sparse array, then `Y` is also returned as a sparse array.

See also

SPQRFactor, *spqr_factor*, *spqr*, *spqr_solve*, *spqr_qmult*

Notes

This function is part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

solve

SPQRFactor.**solve**(*b*, ***, *transpose=False*, *rhs_batch_size=100*)

Solve a linear system using the SPQR factorization.

Solve a linear system for x given the right-hand side b as either a vector or a matrix with multiple right-hand sides.

If `transpose=False`, solve

$$Ax = b$$

or, if `transpose=True`, solve

$$A^\top x = b.$$

Parameters

- **b** (*(M,) or (M, K) numpy.ndarray*) – The right-hand side vector or matrix. `M` should be the number of rows in `A` if `transpose=False`, otherwise the number of columns.

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

- **transpose** (*bool, optional*) – Whether to solve the transposed system. Default is False.
- **rhs_batch_size** (*int, optional*) – If `b` is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix. If `b` is a 1D array, then `x` is returned as a 1D array. If `b` is a 2D array with `K` columns, then `x` is returned as a 2D array with `K` columns. If `b` is a sparse array, then `x` is also returned as a sparse array. `N` is the number of columns in `A` if `transpose=False`, otherwise the number of rows.

See also

SPQRFactor, *spqr_factor*, *spqr*, *spqr_qmult*

Notes

Part of an interface to the SuiteSparse SPQR library[1].

Added in version 0.5.0.

References

SPQRInfo

class sksparse.spqr.**SPQRInfo**

A dataclass to hold SPQR info statistics.

nnzR_upper_bound

Bound on the number of nonzeros in `R`.

Type
int

nnzH_upper_bound

Bound on the number of nonzeros in `H`.

Type
int

nf

Number of frontal matrices.

Type
int

rank_A_estimate

Estimated rank of `A`.

Type
int

n1cols

Number of singleton columns.

[1] SuiteSparse SPQR https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/SPQR

Type
int

`n1rows`
Number of singleton rows.

Type
int

`ordering`
Ordering method used.

Type
str

`memory`
Memory usage in bytes.

Type
int

`flops_upper_bound`
Upper bound on flop count (excluding backsolve).

Type
int

`tol`
Column norm tolerance used.

Type
float

`norm_E_fro`
Norm of dropped diagonal of R.

Type
float

`analyze_time`
Time taken for the symbolic analysis in seconds.

Type
float

`factorize_time`
Time take for the numeric factorization (including applying `Q.T`)

Type
int

`solve_time`
Time taken for the backsolve only $Rx = Q^T b$ in seconds.

Type
int

`total_time`
Total time in seconds.

Type
int

flops
Actual flops for the factorization and solve (including backsolve).

Type
int

Added in version 0.5.0.

Methods

__init__

__init__

SPQRInfo.**__init__**(**args*, ***kwargs*)

Warnings and Exceptions

SPQRWarning	Base class for SPQR warnings.
SPQRRankDeficiencyWarning	Raised when SPQR detects a rank-deficient matrix.
SPQRError	Base class for SPQR exceptions.
SPQRNotInstalledError	Raised when the SPQR library is not installed.
SPQROutOfMemoryError	Raised when SPQR runs out of memory.
SPQROverflowError	Raised when SPQR encounters an integer overflow.
SPQRInvalidInputError	Raised when SPQR receives invalid input.
SPQRGpuProblemError	Raised when SPQR encounters a problem with CUDA.

SPQRWarning

exception sksparse.spqr.**SPQRWarning**
Bases: Warning

Base class for SPQR warnings.

SPQRRankDeficiencyWarning

exception sksparse.spqr.**SPQRRankDeficiencyWarning**
Bases: *SPQRWarning*

Raised when SPQR detects a rank-deficient matrix.

SPQRError

exception sksparse.spqr.**SPQRError**
Bases: Exception

Base class for SPQR exceptions.

SPQRNotInstalledError

exception sksparse.spqr.**SPQRNotInstalledError**
Bases: *SPQRError*

Raised when the SPQR library is not installed.

SPQROutOfMemoryError

exception sksparse.spqr.**SPQROutOfMemoryError**

Bases: MemoryError, *SPQRError*

Raised when SPQR runs out of memory.

SPQROverflowError

exception sksparse.spqr.**SPQROverflowError**

Bases: *SPQRError*

Raised when SPQR encounters an integer overflow.

SPQRInvalidInputError

exception sksparse.spqr.**SPQRInvalidInputError**

Bases: *SPQRError*

Raised when SPQR receives invalid input.

SPQRGpuProblemError

exception sksparse.spqr.**SPQRGpuProblemError**

Bases: *SPQRError*

Raised when SPQR encounters a problem with CUDA.

References

- SuiteSparse homepage
- SuiteSparse SPQR

4.3.9 Unsymmetric Multifrontal LU Decomposition (*sksparse.umfpack*)

Added in version 0.5.0.

An interface to the SuiteSparse UMFPACK package, which computes the LU factorization and solves systems of equations for sparse, possibly non-symmetric, indefinite matrices.

Function Interface

umf_solve	Solve a linear system using UMFPACK.

umf_solve

sksparse.umfpack.**umf_solve**(*A*, *b*, ***, *trans='N'*, *rhs_batch_size=100*, *control=None*, ***kwargs*)

Solve a linear system using UMFPACK.

This is a convenience function that creates a *UMFFactor* object, computes the numeric factorization, and solves the linear system.

Parameters

- **A** (*(N, N) numpy.ndarray or sparse array*) – The input matrix.

- **b** (*(N,) or (N, K) numpy.ndarray or sparse array*) – The right-hand side vector or matrix.
- **trans** (*str, optional*) – The type of system to solve. Possible values are:
  - `N`: solve $Ax = b$ (default)
  - `T`: solve $A^\top x = b$
  - `H`: solve $A^H x = b$

  Note

  If A is real, then `T` and `H` are equivalent.

- **rhs_batch_size** (*int, optional*) – If `b` is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.
- **control** (*UMFControl*, optional) – The control parameters to use for the factorization. If not provided, default parameters are used.
- **kwargs** (*keyword arguments, optional*) – Additional keyword arguments to pass to *UMFControl* if `control` is not provided.

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix of the same type and shape as the input right-hand side `b`.

Raises

UMFPACKSingularMatrixWarning – If the matrix is detected to be singular to working precision. In that case, the solution will have infinite or NaN values, but other entries may still be valid.

See also

UMFFactor, *UMFControl*, *umf_factor*

Added in version 0.5.0.

Notes

The underlying UMFPACK solver can only handle 1D dense array inputs. If the RHS `b` is a 2D array, this method will solve each column independently. If `b` is dense, there is a slight performance gain (~5% in time) by passing it as a Fortran-contiguous array (*e.g.* by using `numpy.asfortranarray()`), since the columns are then stored contiguously in memory. Otherwise, each column will be copied to a temporary Fortran-contiguous buffer before solving.

Examples

See: Davis, Timothy A. (2006). Direct Methods for Sparse Linear Systems, p 74 (Figure 5.1)

```
>>> import numpy as np
>>> from scipy import sparse
>>> from sksparse.umfpack import umf_solve
>>> N = 8
>>> rows = np.array(
```

(continues on next page)

(continued from previous page)

```
... [0, 1, 2, 3, 4, 5, 6, 3, 6, 1, 6, 0, 2, 5, 7, 4, 7, 0, 1, 3, 7, 5, 6],
... dtype=np.int32,
...)
>>> cols = np.array(
... [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7],
... dtype=np.int32,
...)
>>> vals = np.ones(len(rows), dtype=np.float64)
>>> vals[:7] = np.arange(1, 8, dtype=np.float64)  # make diagonal entries non-unit
>>> A = sparse.csc_array((vals, (rows, cols)), shape=(N, N))
>>> A
<Compressed Sparse Column sparse array of dtype 'float64'
        with 23 stored elements and shape (8, 8)>
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = umf_solve(A, b)
>>> np.allclose(x, expect_x)
True
```

Object Interface

umf_factor	Compute the LU factorization of a sparse matrix using UMFPACK.
UMFFactor	The main object used for creating and using an LU factorization.
UMFInfo	A data class to store UMFPACK info.
UMFControl	The class used to manage UMFPACK control parameters.

umf_factor

sksparse.umfpack.**umf_factor**(*A*, *, *control=None*, ***kwargs*)

Compute the LU factorization of a sparse matrix using UMFPACK.

This is a convenience function that creates a *UMFFactor* object, computes the numeric factorization, and returns the resulting object.

Parameters

- **A** (*(M, N) numpy.ndarray or sparse array*) – The input matrix to factorize.
- **control** (*UMFControl*, optional) – The control parameters to use for the factorization. If not provided, default parameters are used.
- **kwargs** (*keyword arguments, optional*) – Additional keyword arguments to pass to *UMFControl* if `control` is not provided.

Returns

UMFFactor – The LU factorization of the input matrix.

Raises

UMFPACKSingularMatrixWarning – If the matrix is exactly singular.

See also

UMFFactor, *UMFControl*, *umf_solve*

Added in version 0.5.0.

Examples

See: Davis, Timothy A. (2006). Direct Methods for Sparse Linear Systems, p 74 (Figure 5.1)

```
>>> import numpy as np
>>> from scipy import sparse
>>> from sksparse.umfpack import umf_factor
>>> N = 8
>>> rows = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 3, 6, 1, 6, 0, 2, 5, 7, 4, 7, 0, 1, 3, 7, 5, 6],
...    dtype=np.int32,
...)
>>> cols = np.array(
...    [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7],
...    dtype=np.int32,
...)
>>> vals = np.ones(len(rows), dtype=np.float64)
>>> vals[:7] = np.arange(1, 8, dtype=np.float64)  # make diagonal entries non-unit
>>> A = sparse.csc_array((vals, (rows, cols)), shape=(N, N))
>>> A
<Compressed Sparse Column sparse array of dtype 'float64'
        with 23 stored elements and shape (8, 8)>
>>> # Compute the LU factorization
>>> f = umf_factor(A)
>>> f
<UMFFactor numeric factor of dtype 'float64' with 'int32' indices:
    L: (8, 8) with 15 stored elements
    U: (8, 8) with 16 stored elements>
>>> L, U, p, q, r = f  # unpack the factorization
>>> LU = (L @ U).toarray()
>>> PRAQ = (r[:, np.newaxis] * A).tocsc()[p][:, q].toarray()
>>> np.allclose(LU, PRAQ)
True
>>> # Solve a linear system
>>> expect_x = np.arange(N, dtype=np.float64)
>>> b = A @ expect_x
>>> x = f.solve(b)
>>> np.allclose(x, expect_x)
True
```

UMFFactor

class sksparse.umfpack.**UMFFactor**

The main object used for creating and using an LU factorization.

The constructor computes the symbolic analysis of a sparse matrix A and determines a fill-reducing ordering

such that:

$$LU = PRAQ.$$

The numeric factorization is not computed until *factorize()* is called.

Note

Note that the use of the scale factor R differs between KLU and UMFPACK:

$$LU = PR_{\text{umf}}AQ$$
(UMFPACK),
$$LU + F = R_{\text{klu}}PAQ$$
(KLU).

They are related by $R_{\text{klu}} = PR_{\text{umf}}P^{\top}$.

is_numeric
Whether the numeric factorization has been computed.

Type
bool

lnz, unz
Number of nonzeros in L and U, respectively.

Type
int

nnz
Total number of nonzeros in L and U.

Type
int

n_row, n_col
Number of rows and columns in the input matrix.

Type
int

nz_udiag
Number of nonzeros on the diagonal of U.

Type
int

dtype
The data type of the matrix entries (`float64` or `complex128`).

Type
numpy.dtype

itype
The integer type used for indexing (`int32` or `int64`).

Type
numpy.dtype

L

The L factor as a sparse CSR matrix.

Type
scipy.sparse.csr_array

U

The U factor as a sparse CSC matrix.

Type
scipy.sparse.csc_array

perm_r, perm_c

The row and column permutation arrays, P and Q.

Type
numpy.ndarray

R

The row scaling diagonal matrix as a 1D array.

Type
numpy.ndarray

info

An object containing information about the factorization.

Type
UMFInfo

control

An object containing settings for the factorization.

Type
UMFControl

See also

UMFControl, *umf_factor*, *umf_solve*

Notes

This object is an interface to the SuiteSparse UMFPACK library[1].

Added in version 0.5.0.

References

Methods

__init__	Compute the symbolic analysis.
copy	Return a deep copy of the current UMFFactor object.
factorize	Compute the numeric factorization of a sparse matrix.
report_control	Print a report of the control structure to stdout.
report_info	Print a report of the UMFInfo structure.

continues on next page

[1] SuiteSparse UMFPACK https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/UMFPACK

Table 35 – continued from previous page

report_numeric	Print a report of the numeric factorization.
report_symbolic	Print a report of the symbolic factorization.
slogdet	Return the determinant of the matrix as (sign, logabsdet).
solve	Solve a linear system using the LU factorization.

__init__

UMFFactor.__init__()

Compute the symbolic analysis.

Parameters

- **A** (*numpy.ndarray or sparse array*) – The input matrix. Any object that can be converted to a csc_array is accepted.
- **control** (*UMFControl*, optional) – An object containing settings for the factorization. Default values will be used if not provided.

copy

UMFFactor.copy()

Return a deep copy of the current UMFFactor object.

factorize

UMFFactor.factorize(*A=None*)

Compute the numeric factorization of a sparse matrix.

Given the symbolic analysis performed in the constructor, compute the numeric factorization of a sparse matrix A and determine a fill-reducing ordering such that:

$$LU = PRAQ.$$

If given, the matrix A must have the same shape and nonzero pattern as the one used to create this *UMFFactor* object, but need not have the same values.

Parameters

A (*(M, N) numpy.ndarray or sparse array*) – The input matrix. Must have the same shape and nonzero pattern as the matrix used to create this *UMFFactor* object. If not provided, the original matrix given to the constructor will be used.

Returns

UMFFactor – The current object, for method chaining.

report_control

UMFFactor.report_control()

Print a report of the control structure to stdout.

This method provides more internal details from UMFPACK itself than the string representation.

> **Note**
>
> This method temporarily sets the print level to 2 (print all information) and restores the previous value afterwards, so the report will *always* show "print level: 2". Use `UMFControl.print_level` or `print(UMFControl)` to see the actual print level.

report_info

`UMFFactor.`**`report_info`**(*print_level=2*)

Print a report of the UMFInfo structure.

This method provides more internal details from UMFPACK itself than the string representation.

Parameters

print_level (*int, optional*) – The verbosity level. Default value is 2.

Accepted values are:

- None: use current print level
- <= 0: no output
- 1: error messages only
- >= 2: error messages and print all of UMFInfo

report_numeric

`UMFFactor.`**`report_numeric`**(*print_level=4*)

Print a report of the numeric factorization.

This method provides more internal details from UMFPACK itself than the string representation.

Parameters

print_level (*int, optional*) – The verbosity level. Default value is 4.

Accepted values are:

- None: use current print level
- <= 2: no printing
- 3: fully check input, and print a short summary of its status
- 4: as 3, but print first few entries of the input
- 5: as 3, but print all of the input

report_symbolic

`UMFFactor.`**`report_symbolic`**(*print_level=4*)

Print a report of the symbolic factorization.

This method provides more internal details from UMFPACK itself than the string representation.

Parameters

print_level (*int, optional*) – The verbosity level. Default value is 4.

Accepted values are:

- None: use current print level

- <= 2: no printing
- 3: fully check input, and print a short summary of its status
- 4: as 3, but print first few entries of the input
- 5: as 3, but print all of the input

slogdet

`UMFFactor.slogdet()`

Return the determinant of the matrix as (sign, logabsdet).

See also

`numpy.linalg.slogdet`

solve

`UMFFactor.solve(b, *, trans='N', rhs_batch_size=100)`

Solve a linear system using the LU factorization.

This method solves one of the following linear systems:

- $Ax = b$ (if `trans='N'`)
- $A^\top x = b$ (if `trans='T'` and A is real)
- $A^H x = b$ (if `trans='H'` and A is complex)

Parameters

- **b** (*(N,) or (N, K) numpy.ndarray or sparse array*) – The right-hand side vector or martrix.
- **trans** (*str, optional*) – The type of system to solve. Possible values are:
  - `N`: solve $Ax = b$ (default)
  - `T`: solve $A^\top x = b$
  - `H`: solve $A^H x = b$

  Note

  If A is real, then `T` and `H` are equivalent.

- **rhs_batch_size** (*int, optional*) – If `b` is a 2D sparse array, this parameter controls the number of columns to be solved simultaneously. A larger number will increase memory consumption by converting more columns at a time to dense arrays, but may improve runtime.

Returns

x (*(N,) or (N, K) numpy.ndarray or sparse array*) – The solution vector or matrix. If `b` is a 1D array, then `x` is returned as a 1D array. If `b` is a 2D array with `K` columns, then `x` is returned as a 2D array with `K` columns. If `b` is a sparse array, then `x` is also returned as a sparse array.

Raises

UMFPACKSingularMatrixWarning – If the matrix is detected to be singular to working precision. In that case, the solution will have infinite or NaN values, but other entries may still be valid.

Notes

The underlying UMFPACK solver can only handle 1D dense array inputs. If the RHS `b` is a 2D array, this method will solve each column independently. If `b` is dense, there is a slight performance gain (~5% in time) by passing it as a Fortran-contiguous array (*e.g.* by using `numpy.asfortranarray()`), since the columns are then stored contiguously in memory. Otherwise, each column will be copied to a temporary Fortran-contiguous buffer before solving.

UMFInfo

class sksparse.umfpack.**UMFInfo**

A data class to store UMFPACK info.

status

Return status of the last UMFPACK call.

Type
int

n_row

Number of rows in the input matrix.

Type
int

n_col

Number of columns in the input matrix.

Type
int

nz

Number of nonzeros in the input matrix.

Type
int

size_of_unit

Size of a unit in bytes.

Type
int

size_of_int

Size of an *int32_t* in bytes.

Type
int

size_of_long

Size of an *int64_t* in bytes.

Type
int

size_of_pointer

Size of a *void* * pointer in bytes.

Type
int

`size_of_entry`

Size of an entry in bytes, real or complex.

Type
int

`ndense_row`

Number of dense rows in the input matrix.

Type
int

`nempty_row`

Number of empty rows in the input matrix.

Type
int

`ndense_col`

Number of dense columns in the input matrix.

Type
int

`nempty_col`

Number of empty columns in the input matrix.

Type
int

`symbolic_defrag`

Number of memory compactions performed.

Type
int

`symbolic_peak_memory`

Peak memory usage during symbolic factorization.

Type
int

`symbolic_size`

Size of symbolic factorization, in Units.

Type
int

`symbolic_time`

Time spent in symbolic factorization, in seconds.

Type
float

`symbolic_walltime`

Wall-clock time spent in symbolic factorization, in seconds.

Type
float

`strategy_used`

Strategy used in the factorization. One of: `{"auto", "unsymmetric", "symmetric"}`.

Type
str

`ordering_used`

Ordering method used in the factorization. One of: `{"cholmod", "amd", "given", "none", "metis", "best", "user", "metis_guard"}`

Type
str

`qfixed`

Whether the column permutation Q was fixed.

Type
bool

`diag_preferred`

Whether diagonal pivoting was preferred.

Type
bool

`pattern_symmetry`

Symmetry of the nonzero pattern of the input matrix, excluding dense rows and columns (aka S).

Type
float

`nz_a_plus_at`

Number of nonzeros in $S + S^\top$, excluding the diagonal.

Type
int

`nzdiag`

Number of nonzeros on the diagonal of S.

Type
int

`symmetric_lunz`

Number of non-zeros in $L + U$, if AMD ordering was used.

Type
int

`symmetric_flops`

Number of floating-point operations for the factorization, if AMD ordering was used.

Type
int

`symmetric_ndense`

Number of dense rows and columns in $S + S^\top$.

Type
int

`symmetric_dmax`

Maximum number of entries in any column of L, for AMD.

Type
int

`col_singletons`

Number of column singletons.

Type
int

`row_singletons`

Number of row singletons.

Type
int

`n2`

Size of S.

Type
int

`s_symmetric`

1 if S is square and symmetrically permuted.

Type
int

`numeric_size_estimate`

Estimated size of numeric factorization, in Units.

Type
int

`peak_memory_estimate`

Estimated peak memory usage during numeric factorization.

Type
int

`flops_estimate`

Estimated number of floating-point operations for the factorization.

Type
int

`lnz_estimate`

Estimated number of nonzeros in L.

Type
int

`unz_estimate`

Estimated number of nonzeros in U.

Type
int

`variable_init_estimate`
Initial size of memory usage in numeric factorization.

Type
int

`variable_peak_estimate`
Peak size of memory usage in numeric factorization.

Type
int

`variable_final_estimate`
Final size of memory usage in numeric factorization.

Type
int

`max_front_size_estimate`
Maximum frontal matrix size, estimated.

Type
int

`max_front_nrows_estimate`
Maximum number of rows in any frontal matrix, estimated.

Type
int

`max_front_ncols_estimate`
Maximum number of columns in any frontal matrix, estimated.

Type
int

`numeric_size`
Size of numeric factorization, in Units.

Type
int

`peak_memory`
Peak memory usage during symbolic and numeric factorization.

Type
int

`flops`
Number of floating-point operations for the factorization.

Type
int

`lnz`
Number of nonzeros in L.

Type
int

unz

Number of nonzeros in U.

Type
int

variable_init

Initial size of memory usage in numeric factorization.

Type
int

variable_peak

Peak size of memory usage in numeric factorization.

Type
int

variable_final

Final size of memory usage in numeric factorization.

Type
int

max_front_size

Maximum frontal matrix size.

Type
int

max_front_nrows

Maximum number of rows in any frontal matrix.

Type
int

max_front_ncols

Maximum number of columns in any frontal matrix.

Type
int

numeric_defrag

Number of memory compactions performed.

Type
int

numeric_realloc

Number of memory reallocations performed.

Type
int

numeric_costly_realloc

Number of costly memory reallocations performed.

Type
int

`compressed_pattern`

Number of integers in LU pattern.

Type
int

`lu_entries`

Number of real entries in L and U.

Type
int

`numeric_time`

Time spent in numeric factorization, in seconds.

Type
float

`nz_udiag`

Number of nonzeros on the diagonal of U.

Type
int

`rcond`

Estimate of the reciprocal of the condition number of A.

Type
float

`was_scaled`

Scaling method used. One of: `{"none", "sum", "max"}`.

Type
str

`rsmin`

min(max(row)) or *min(sum(row))*, depending on the scaling method.

Type
float

`rsmax`

max(max(row)) or *max(sum(row))*, depending on the scaling method.

Type
float

`umin`

Minimum absolute value of a diagonal entry of U.

Type
float

`umax`

Maximum absolute value of a diagonal entry of U.

Type
float

`alloc_init_used`

Initial memory allocation used, as a fraction of total numeric memory.

Type
float

`forced_updates`

Number of forced updates during numeric factorization.

Type
int

`numeric_walltime`

Wall-clock time spent in numeric factorization, in seconds.

Type
float

`noff_diag`

Number of off-diagonal pivots.

Type
int

`all_lnz`

Total number of entries in L, if no dropped entries.

Type
int

`all_unz`

Total number of entries in U, if no dropped entries.

Type
int

`nzdropped`

Number of dropped entries in L and U.

Type
int

`ir_taken`

Number of iterative refinement steps taken.

Type
int

`ir_attempted`

Number of iterative refinement steps attempted.

Type
int

`omega1`

Factor for sparse backdward error estimate.

Type
int

omega2
Factor for sparse backdward error estimate.

Type
int

solve_flops
Number of floating-point operations for *solve*.

Type
int

solve_time
Time spent in *solve*, in seconds.

Type
float

solve_walltime
Wall-clock time spent in *solve*, in seconds.

Type
float

Added in version 0.5.0.

Methods

__init__

__init__

UMFInfo.**__init__**()

UMFControl

class sksparse.umfpack.**UMFControl**
The class used to manage UMFPACK control parameters.

print_level
The verbosity level. Values vary depending on the function called, but typically "0" means no printing, and higher values mean more verbose printing. Default value is 1.

Type
int

dense_row, dense_col
A row or column is considered to be dense if it has more than `max(16, dense_[row|col] * 16 * sqrt(n_[row|col])` entries. Default 0.2.

Type
int

blas3_block_size
The block size to use in Level-3 BLAS operations. Default value is 32.

Type
int

strategy

The strategy to use in the factorization. Default value is `auto`. Possible values are:

- `auto`: choose the strategy automatically
- `unsymmetric`: order the columns of A with COLAMD
- `symmetric`: Order the matrix $A + A^\top$ with AMD

Type
str

ordering_method

The ordering method to use. Default value is `amd`. Possible values are:

- `cholmod`: use AMD/COLAMD, then METIS
- `amd`: just use AMD or COLAMD
- `given`: use the user-provided ordering
- `none`: no ordering
- `metis`: use METIS on $A + A^\top$ or $A^\top A$
- `best`: try AMD/COLAMD, METIS and NESDIS
- `user`: use the user-provided function to compute the ordering
- `metis_guard`: use METIS for symmetric strategy, try METIS for unsymmetric and fall back to COLAMD if A has many dense rows.

Type
str

fixQ

Default 0. Possible values:

- -1: possibly modify Q during numeric factorization.
- 0: automatic. Modify Q only if strategy is unsymmetric.
- 1: do not modify Q during numeric factorization.

Type
int

amd_dense

Rows/columns in $A + A^\top$ with more than `max(16, amd_dense * sqrt(n))` entries (where `n = n_row = n_col`) are ignored in the AMD pre-ordering. Default 10.

Type
int

aggressive

If True, use aggressive absorption in AMD. Default True.

Type
bool

`singletons`
: If True, remove singletons prior to factorization. Default True.

  Type
  : bool

`pivot_tol`
: The relative pivot tolerance for partial pivoting with row interchanges. The absolute value of the entry must be `>= pivot_tol *` largest absolute value in that column. `pivot_tol=1.0` gives true partial pivoting. If `pivot_tol <= 0.0`, then any non-zero entry is acceptable as a pivot. Default value is 0.1.

  Type
  : float

`sym_pivot_tol`
: The relative pivot tolerance for symmetric strategy. Default 0.001.

  Type
  : float

`row_scale`
: The row scaling to use. Default value is `'sum'`. Possible values are:

  - None or `'none'`: no row scaling
  - `'sum'`: divide each row by `sum(abs(A[i,:]))`
  - `'max'`: divide each row by `max(abs(A[i,:]))`

  Type
  : str or None

`alloc_init`
: Estimated space for the memory to allocate for numeric factorization. Default 0.7.

  Type
  : float

`front_alloc_init`
: Estimated space for the memory to allocate for frontal matrices. Default 0.5.

  Type
  : float

`droptol`
: Drop tolerance for small entries in L and U. Default value is 0.0 (no dropping).

  Type
  : float

`ir_steps`
: Number of iterative refinement steps to perform. Default value is 2.

  Type
  : int

`compiles_with_blas`
: True if UMFPACK was compiled with BLAS support. Read-only.

  Type
  : bool

sym_thresh

Threshold for choosing symmetric strategy. Default 0.3.

Type
float

nnzdiag_thresh

Threshold for choosing unsymmetric strategy based on the number of diagonal entries. Default 0.9.

Type
float

Added in version 0.5.0.

Methods

__init__	
report	Print a report of the control structure to stdout.

__init__

UMFControl.**__init__**()

report

UMFControl.**report**()

Print a report of the control structure to stdout.

This method provides more internal details from UMFPACK itself than the string representation.

> **Note**
>
> This method temporarily sets the print level to 2 (print all information) and restores the previous value afterwards, so the report will *always* show "print level: 2". Use `UMFControl.print_level` or `print(UMFControl)` to see the actual print level.

Warnings and Exceptions

UMFPACKWarning	A warning occurred in a UMFPACK routine.
UMFPACKSingularMatrixWarning	A singular matrix was encountered in a UMFPACK routine.
UMFPACKDeterminantUnderflowWarning	A determinant underflow was encountered in a UMFPACK routine.
UMFPACKDeterminantOverflowWarning	A determinant overflow was encountered in a UMFPACK routine.
UMFPACKError	An error occurred in a UMFPACK routine.
UMFPACKOutOfMemoryError	UMFPACK ran out of memory.
UMFPACKInvalidNumericObjectError	An invalid Numeric object was passed to a UMFPACK routine.
UMFPACKInvalidSymbolicObjectError	An invalid Symbolic object was passed to a UMFPACK routine.

continues on next page

Table 38 – continued from previous page

UMFPACKArgumentMissingError	A required argument was missing in a UMFPACK routine.
UMFPACKNonpositiveError	A non-positive value for n was passed to a UMFPACK routine.
UMFPACKInvalidMatrixError	An invalid matrix was passed to a UMFPACK routine.
UMFPACKDifferentPatternError	A matrix with a different nonzero pattern was passed to a UMFPACK routine.
UMFPACKInvalidSystemError	An invalid system type was passed to a UMFPACK routine.
UMFPACKInvalidPermutationError	An invalid permutation was passed to a UMFPACK routine.
UMFPACKInternalError	An internal error occurred in a UMFPACK routine.
UMFPACKFileIOError	A file I/O error occurred in a UMFPACK routine.
UMFPACKOrderingFailedError	The ordering algorithm failed in a UMFPACK routine.
UMFPACKInvalidBlobError	An invalid blob was passed to a UMFPACK routine.
UMFPACKSingularMatrixError	A singular matrix was encountered in a UMFPACK routine.

UMFPACKWarning

exception sksparse.umfpack.**UMFPACKWarning**

Bases: Warning

A warning occurred in a UMFPACK routine.

UMFPACKSingularMatrixWarning

exception sksparse.umfpack.**UMFPACKSingularMatrixWarning**

Bases: *UMFPACKWarning*

A singular matrix was encountered in a UMFPACK routine.

UMFPACKDeterminantUnderflowWarning

exception sksparse.umfpack.**UMFPACKDeterminantUnderflowWarning**

Bases: *UMFPACKWarning*

A determinant underflow was encountered in a UMFPACK routine.

UMFPACKDeterminantOverflowWarning

exception sksparse.umfpack.**UMFPACKDeterminantOverflowWarning**

Bases: *UMFPACKWarning*

A determinant overflow was encountered in a UMFPACK routine.

UMFPACKError

exception sksparse.umfpack.**UMFPACKError**

Bases: Exception

An error occurred in a UMFPACK routine.

UMFPACKOutOfMemoryError

exception sksparse.umfpack.**UMFPACKOutOfMemoryError**

Bases: MemoryError, *UMFPACKError*

UMFPACK ran out of memory.

UMFPACKInvalidNumericObjectError

exception sksparse.umfpack.**UMFPACKInvalidNumericObjectError**

Bases: *UMFPACKError*

An invalid Numeric object was passed to a UMFPACK routine.

UMFPACKInvalidSymbolicObjectError

exception sksparse.umfpack.**UMFPACKInvalidSymbolicObjectError**

Bases: *UMFPACKError*

An invalid Symbolic object was passed to a UMFPACK routine.

UMFPACKArgumentMissingError

exception sksparse.umfpack.**UMFPACKArgumentMissingError**

Bases: *UMFPACKError*

A required argument was missing in a UMFPACK routine.

UMFPACKNonpositiveError

exception sksparse.umfpack.**UMFPACKNonpositiveError**

Bases: *UMFPACKError*

A non-positive value for n was passed to a UMFPACK routine.

UMFPACKInvalidMatrixError

exception sksparse.umfpack.**UMFPACKInvalidMatrixError**

Bases: *UMFPACKError*

An invalid matrix was passed to a UMFPACK routine.

UMFPACKDifferentPatternError

exception sksparse.umfpack.**UMFPACKDifferentPatternError**

Bases: *UMFPACKError*

A matrix with a different nonzero pattern was passed to a UMFPACK routine.

UMFPACKInvalidSystemError

exception sksparse.umfpack.**UMFPACKInvalidSystemError**

Bases: *UMFPACKError*

An invalid system type was passed to a UMFPACK routine.

UMFPACKInvalidPermutationError

exception sksparse.umfpack.**UMFPACKInvalidPermutationError**

Bases: *UMFPACKError*

An invalid permutation was passed to a UMFPACK routine.

UMFPACKInternalError

exception sksparse.umfpack.**UMFPACKInternalError**

Bases: *UMFPACKError*

An internal error occurred in a UMFPACK routine.

UMFPACKFileIOError

exception sksparse.umfpack.**UMFPACKFileIOError**

Bases: *UMFPACKError*

A file I/O error occurred in a UMFPACK routine.

UMFPACKOrderingFailedError

exception sksparse.umfpack.**UMFPACKOrderingFailedError**

Bases: *UMFPACKError*

The ordering algorithm failed in a UMFPACK routine.

UMFPACKInvalidBlobError

exception sksparse.umfpack.**UMFPACKInvalidBlobError**

Bases: *UMFPACKError*

An invalid blob was passed to a UMFPACK routine.

UMFPACKSingularMatrixError

exception sksparse.umfpack.**UMFPACKSingularMatrixError**

Bases: *UMFPACKError*

A singular matrix was encountered in a UMFPACK routine.

References

- SuiteSparse homepage
- SuiteSparse UMFPACK

Provides sparse matrix algorithms not found in SciPy, for use with SciPy's sparse matrix classes in `scipy.sparse`.

4.3.10 Submodules

amd	Approximate Minimum Degree (AMD) Ordering (sksparse.amd)
btf	Block Triangular Form (BTF) (sksparse.btf)

continues on next page

Table 39 – continued from previous page

camd	Constrained Approximate Minimum Degree (CAMD) Ordering (sksparse.camd)
ccolamd	Constrained Column Approximate Minimum Degree (CCOLAMD) Ordering (sksparse.ccolamd)
cholmod	Cholesky Decomposition (sksparse.cholmod)
colamd	Column Approximate Minimum Degree (COLAMD) Ordering (sksparse.colamd)
klu	Clark Kent LU Decomposition (sksparse.klu)
spqr	Sparse QR Decomposition (sksparse.spqr)
umfpack	Unsymmetric Multifrontal LU Decomposition (sksparse.umfpack)

References

- SuiteSparse homepage
- SuiteSparse GitHub

4.4 Changes

4.4.1 v0.5.0

Upgrading from v0.4.x

The *sksparse.cholmod* module has undergone major API changes in this release. Please see the *API Changes* section below for a detailed list of changes. The new API is not backward compatible with previous versions, so code written for v0.4.x will need to be updated to work with v0.5.0. Code can be upgraded using the following changes to the `sksparse.cholmod` API:

API Change	v0.4.x	v0.5.0
Symbolic Analysis	`analyze(A)`	`CholeskyFactor(A)`
	`analyze_AAt(A)`	`CholeskyFactor(A, sym_kind="row")`
Numeric Factorization	`cholesky(A)`	`cho_factor(A)`
	`cholesky_AAt(A)`	`cho_factor(A, sym_kind="row")`
kwargs	`mode`	`supernodal_mode`
	`ordering_method`	`order`
	`use_long`	not used
Solving Linear Systems	`f.solve_A(b)`	`f.solve(b)`
	`f.solve_LDLt(b)`	`f.solve(b, system="LDLt")`
	`f.solve_LD(b)`	`f.solve(b, system="LD")`
	`f.solve_DLt(b)`	`f.solve(b, system="DLt")`
	`f.solve_L(b)`	`f.solve(b, system="L")`
	`f.solve_Lt(b)`	`f.solve(b, system="Lt")`
	`f.solve_D(b)`	`f.solve(b, system="D")`

continues on next page

Table 40 – continued from previous page

API Change	v0.4.x	v0.5.0
Modifying Factorization	`f.cholesky(A)`	`f = cho_factor(A)`
	`f.cholesky_AAt(A)`	`f = cho_factor(A, sym_kind="row")`
	`f.cholesky_inplace(A)`	`f.factorize(A)`
	`f.cholesky_AAt_inplace(A)`	`f.factorize(A, sym_kind="row")`
	`f.update_inplace(C, subtract=False)`	`f.update(C)`
	`f.update_inplace(C, subtract=True)`	`f.downdate(C)`
Extracting Factors	`L = f.L()`	`L = f.get_factor(kind="LL", lower=True)`

		`if not f.is_ll:`
		`  f.change_factor(kind=`
		`↪"LL")`
		`L = f.L`

	`LD = f.LD()`	`if f.is_ll:`
		`  f.change_factor(kind=`
		`↪"LDL")`
		`LD = scipy.sparse.csc_`
		`↪array(f.factor)`

	`L, D = f.L_D()`	`L, D = f.get_factor(kind="LDL", lower=True)`

		`if f.is_ll:`
		`  f.change_factor(kind=`
		`↪"LDL")`
		`L, D = f.L, f.D`

	`D = f.D()`	`_, D = f.get_factor(kind="LDL", lower=True)`

		`if f.is_ll:`
		`  f.change_factor(kind=`
		`↪"LDL")`
		`D = f.D`

	`p = f.P()`	`p = f.get_perm()`
Permuting Vectors	`f.apply_P(b)`	`b[f.perm]`
	`f.apply_Pt(b)`	`b[np.argsort(f.perm)]`

API Changes

- Major API updates to the *sksparse.cholmod* module. The module has been updated to resemble the existing `scipy.linalg.cholesky()` interface, as well as provide additional functions present in the SuiteSparse CHOLMOD MATLAB interface, and MATLAB built-in Cholesky functions. The underlying Cython code has been refactored to provide greater type safety and performance.
  - The `cholmod.Factor` class has been renamed to *CholeskyFactor*.

- The `cholmod.Common` class has been removed. Its attributes have been subsumed into the `CholeskyFactor` class.
- The *cholesky()* function now returns a `csc_array` instead of a `Factor` object, and an optional `ndarray` containing the permutation vector.
- The *ldl()* function has been added. It returns a tuple (`csc_array`, `csc_array`), and an optional `ndarray` containing the permutation vector.
- A *cho_factor()* function has been added to perform the numeric Cholesky factorization and return a *CholeskyFactor* object.
- Similarly, a *ldl_factor()* function has been added to perform the numeric LDL factorization and return a *CholeskyFactor* object.
- The `cholmod.analyze` function has been removed. The analysis step is now performed when calling the constructor of *CholeskyFactor*.
- The `use_long` parameter has been removed from the *cholesky()* and *ldl()* functions. The type of indices is now inferred from the input matrix.
- The `mode` parameter has been renamed to `supernodal_mode`.
- The `symmetric` parameter has been removed. It has been replaced by two new parameters: `lower`, and `sym_kind`.
  - Parameter `lower` controls whether to use the lower or upper triangular part of the input matrix, or whether to return a lower or upper triangular factor.
  - Parameter `sym_kind` has been added. It accepts a string argument in `{"sym", "row", "col"}`, which controls the symmetry structure of the matrix to analyze.
- The functions `cholmod.analyze_AAt` and `cholmod.cholesky_AAt` have been removed. Use *cho_factor()* or *cholesky()* with `sym_kind="row"` instead.
- The `ordering_method` parameter has been renamed to `order`.
- The `Factor` methods `L`, `D`, `LD`, `L_D`, and `P`, have been removed in favor of the methods *get_factor()* and *get_perm()*.
- The properties *perm* and *factor* have been added to return read-only views of the permutation vector and factor matrix, respectively.
- The `Factor` methods `solve_A`, `solve_LDLt`, `solve_LD`, `solve_DLt`, `solve_L`, `solve_Lt`, and `solve_D` method have been replaced by the *solve()* method.
- The *CholeskyFactor* is not callable.
- The `Factor.apply_P` and `Factor.apply_Pt` methods have been removed. Use `p = f.get_perm(); x_p = x[p]` and `x = x_p[np.argsort(p)]` instead.
- The new *solve()* method checks the condition number and raises a *CholmodNotPositiveDefiniteError* if the matrix is exactly singular, or a *CholmodWarning* if the matrix is ill-conditioned. Previously, no warning would be issued. See the *rcond* property for more details.
- Add multiple properties to the *CholeskyFactor* class for convenient access to `cholmod_factor` attributes. See the full documentation for details.
- Fix a bug in the previous version where sparse inputs with inconsistent `has_sorted_indices` or `has_canonical_format` flags would silently lead to incorrect results. The input matrix is now modified into a canonical CSC format, regardless of the input format.

  - Add support for single-precision (float32/complex64) input matrices. The output factor and solve results will match the input precision. Previously, all inputs were converted to double-precision.

- Create the `amd` module, which provides the AMD ordering method.
- Create the `btf` module, which provides the BTF ordering method.
- Create the `camd` module, which provides the constrained AMD ordering method.
- Create the `colamd` module, which provides the COLAMD ordering method.
- Create the `ccolamd` module, which provides the constrained COLAMD ordering method.
- Create the `klu` submodule, which provides an interface to the KLU sparse LU solver.
- Create the `spqr` submodule, which provides an interface to the SPQR sparse QR solver.
- Create the `umfpack` submodule, which provides an interface to the UMFPACK sparse LU solver.
- Remove support for the following versions:
  - Python < 3.10
  - NumPy < 2.0
  - SciPy < 1.14
  - SuiteSparse < 7.4.0

  Python 3.9 reached its end of life in October 2025. Numpy ended support for all 1.x versions in September 2025. SciPy v1.14 (released June 2024) will be supported until the end of 2026. SuiteSparse 7.4.0 introduces single precision support in CHOLMOD 5.1.0.

4.4.2 v0.4.4

- Bug in solve with dense array, where base of result is not set correctly, fixed.
- Travis tests are using conda now.
- Supported versions updated to:
  - Python: 3.7, 3.6
  - NumPy: 1.15, 1.14, 1.13
  - SciPy: 1.1, 1.0, 0.19
  - SuiteSparse: 5.2

4.4.3 v0.4.3

- The method `Factor.solve_L` can now also use the L matrix of the LL' decomposition.
- Supported versions updated to:
  - Python: 3.6, 3.5
  - NumPy: 1.14, 1.13
  - SciPy: 1.0, 0.19

4.4.4 v0.4.2

- Bug where the ordering method is not taken into account is fixed.
- The Factor class has now a (public) copy method.

4.4.5 v0.4.1

- Bug with relaxed stride checking in NumPy 1.12 fixed.
- Supported versions updated to:
  - Python: 3.6, 3.5, 3.4, 2.7
  - NumPy: 1.8 to 1.12

4.4.6 v0.4

- 64-bit indices (type long) are now supported.
- The ordering method for Cholesky decomposition is now choosable.
- Specific exceptions subclasses are now thrown for each error condition.
- Setup does not rely on an installed Cython anymore.

4.4.7 v0.3.1

- Ensure that arrays returned by the `Factor.solve_...` methods are writeable.

4.4.8 v0.3

- Dropped deprecated `Factor.solve_P` and `Factor.solve_P`.
- Fixed a memory leak upon garbage collection of `Factor`.

4.4.9 v0.2

- `Factor` solve methods now return 1d output for 1d input (just like `numpy.dot()` does).
- `Factor.solve_P` and `Factor.solve_P` deprecated; use `Factor.apply_P` and `Factor.apply_Pt` instead.
- New methods for computing determinants of positive-definite matrices: `Factor.det`, `Factor.logdet`, `Factor.slogdet`.
- New method for explicitly computing inverse of a positive-definite matrix: `Factor.inv`.
- `Factor.D` has much better implementation.
- Build system improvements.
- Wrapper code re-licensed under BSD terms.

4.4.10 v0.1

First public release.

4.5 Contributing to scikit-sparse

Thank you for your interest in contributing! This project uses a hybrid development environment consisting of Mamba (or Conda, for C-libraries and the Python interpreter) and uv (for Python package management).

4.5.1 Prerequisites

Ensure you have the following installed:

- Mamba (or Miniforge)

4.5.2 Setting Up the Development Environment

We use Mamba to provide the SuiteSparse C-libraries and the C++ compiler.

Create the Environment

From the project root, run:

```
mamba env create -f environment.yml
mamba activate sksparse-dev
```

Install scikit-sparse (Editable)

We use `uv` to install the Python dependencies and compile the Cython extensions. This command installs the project in "editable" mode along with all development tools (pytest, ruff, sphinx).

```
uv pip install -e ".[dev]"
```

> **Note**
>
> If the install fails to find SuiteSparse headers, ensure your `CONDA_PREFIX` is set correctly (`echo $CONDA_PREFIX`). Our `setup.py` is configured to prioritize this path.

4.5.3 Development Workflow

New features, bug fixes, and documentation improvements should be developed in a separate branch off of the dev branch (*e.g.* `feature/my-new-feature`). Once your changes are ready, submit a pull request for review. Follow the NumPy/SciPy contribution guidelines for best practices on documentation and commit messages.

Code Linting and Formatting

We use `ruff` for linting. It is installed as part of the `[dev]` dependencies. It runs automatically on pre-commit, but you can also run it manually:

```
ruff check .
```

Running Tests

We use `pytest` for all unit tests.

```
pytest
```

Rebuilding Extensions

If you modify `.pyx` or `.pxd` files, you must re-run the `uv` install command to trigger a re-compilation of the Cython extensions:

```
uv pip install -e ".[dev]"
```

4.5.4 Building Documentation

The documentation is built using Sphinx and the Furo theme.

```
cd doc
make html
```

The results will be available in `doc/_build/html/index.html`.

4.5.5 Benchmarks

We use asv for benchmarking. See the `./benchmarks/` directory. To run the benchmarks, run the following command from the project root:

```
asv run
```

See also the SciPy benchmark guidelines for best practices on writing benchmarks. We do not use `spin`, so refer to the `asv` commands therein.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W