Chompack Documentation

Release 2.3.1

Martin S. Andersen and Lieven Vandenberghe

February 14, 2017

Contents

Chompack is a library for chordal matrix computations. It includes routines for:

- symbolic factorization
- numeric Cholesky factorization
- forward and back substitution
- maximum determinant positive definite completion
- minimum rank completion
- Euclidean distance matrix completion
- computations with logarithmic barriers for sparse matrix cones
- chordal conversion
- computing a maximal chordal subgraph

The implementation is based on the supernodal-multifrontal algorithms described in these papers:

See also:

L. Vandenberghe and M. S. Andersen, Chordal Graphs and Semidefinite Optimization, *Foundations and Trends in Optimization*, 2015. [doi | bib]

M. S. Andersen, J. Dahl, and L. Vandenberghe, Logarithmic barriers for sparse matrix cones, *Optimization Methods and Software*, 2013. [doi | bib]

Applications of these algorithms in optimization include sparse matrix cone programs, covariance selection, graphical models, and decomposition and relaxation methods.

CHAPTER 1

Copyright and License

CHAPTER 2

Installation

Note: CHOMPACK requires that CVXOPT 1.1.7 or newer is installed.

2.1 Installation from source

The CHOMPACK Python extension can be downloaded, built, and installed by issuing the commands

```
$ git clone https://github.com/cvxopt/chompack.git
$ cd chompack
$ python setup.py install --user
```

Chompack can also be installed using pip

```
$ pip install chompack
```

2.2 Python-only installation

A Python-only reference implementation of CHOMPACK can be installed by setting the environtment variable *CHOMPACK_PY_ONLY=1*

```
$ CHOMPACK_PY_ONLY=1 python install --user
```

or using pip,

```
$ CHOMPACK_PY_ONLY=1 pip install chompack
```

Please note that the Python-only reference implementations are significantly slower than the C implementations that are available in the standard installation.

CHAPTER 3

Documentation

3.1 Quick start

The core functionality of CHOMPACK is contained in two types of objects: the *symbolic* object and the *cspmatrix* (chordal sparse matrix) object. A *symbolic* object represents a symbolic factorization of a sparse symmetric matrix A, and it can be created as follows:

```
from cvxopt import spmatrix, amd
import chompack as cp

# generate sparse matrix
I = [0, 1, 3, 1, 5, 2, 6, 3, 4, 5, 4, 5, 6, 5, 6]
J = [0, 0, 0, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 6]
A = spmatrix(1.0, I, J, (7,7))

# compute symbolic factorization using AMD ordering
symb = cp.symbolic(A, p=amd.order)
```

The argument p is a so-called elimination order, and it can be either an ordering routine or a permutation vector. In the above example we used the "approximate minimum degree" (AMD) ordering routine. Note that A is a lower-triangular sparse matrix that represents a symmetric matrix; upper-triangular entries in A are ignored in the symbolic factorization.

Now let's inspect the sparsity pattern of A and its chordal embedding (i.e., the filled pattern):

```
>>> print(A)
```

```
[ 1.00e+00     0         0         0         0         0         0    ]
[ 1.00e+00  1.00e+00     0         0         0         0         0    ]
[    0         0      1.00e+00     0         0         0         0    ]
[ 1.00e+00     0         0      1.00e+00     0         0         0    ]
[    0         0         0      1.00e+00  1.00e+00     0         0    ]
[    0      1.00e+00     0      1.00e+00  1.00e+00  1.00e+00     0    ]
[    0         0      1.00e+00     0      1.00e+00     0      1.00e+00]
```

```
>>> print(symb.sparsity_pattern(reordered=False, symmetric=False))
```

```
[ 1.00e+00     0         0         0         0         0         0    ]
[ 1.00e+00  1.00e+00     0         0         0         0         0    ]
[    0         0      1.00e+00     0         0         0         0    ]
[ 1.00e+00     0         0      1.00e+00     0         0         0    ]
[    0         0         0      1.00e+00  1.00e+00     0         0    ]
```

```
[ 1.00e+00  1.00e+00     0      1.00e+00  1.00e+00  1.00e+00     0     ]
[    0          0      1.00e+00     0      1.00e+00     0      1.00e+00]
```

The reordered pattern and its cliques can be inspected using the following commands:

```
>>> print(symb)
```

```
[X X           ]
[X X X         ]
[  X X   X X   ]
[      X   X X]
[    X   X X X]
[    X X X X X]
[      X X X X]
```

```
>>> print(symb.cliques())
```

```
[[0, 1], [1, 2], [2, 4, 5], [3, 5, 6], [4, 5, 6]]
```

Similarly, the clique tree, the supernodes, and the separator sets are:

```
>>> print(symb.parent())
```

```
[1, 2, 4, 4, 4]
```

```
>>> print(symb.supernodes())
```

```
[[0], [1], [2], [3], [4, 5, 6]]
```

```
>>> print(symb.separators())
```

```
[[1], [2], [4, 5], [5, 6], []]
```

The *cspmatrix* object represents a chordal sparse matrix, and it contains lower-triangular numerical values as well as a reference to a symbolic factorization that defines the sparsity pattern. Given a *symbolic* object *symb* and a sparse matrix A, we can create a *cspmatrix* as follows:

```
from cvxopt import spmatrix, amd, printing
import chompack as cp
printing.options['dformat'] = '%3.1f'

# generate sparse matrix and compute symbolic factorization
I = [0, 1, 3, 1, 5, 2, 6, 3, 4, 5, 4, 5, 6, 5, 6]
J = [0, 0, 0, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 6]
A = spmatrix([1.0*i for i in range(1,15+1)], I, J, (7,7))
symb = cp.symbolic(A, p=amd.order)

L = cp.cspmatrix(symb)
L += A
```

Now let us take a look at A and L:

```
>>> print(A)
```

```
[ 1.0  0    0    0    0    0    0  ]
[ 2.0  4.0  0    0    0    0    0  ]
[ 0    0    6.0  0    0    0    0  ]
[ 3.0  0    0    8.0  0    0    0  ]
[ 0    0    0    9.0 11.0  0    0  ]
```

```
[ 0     5.0   0    10.0 12.0 14.0   0   ]
[ 0     0     7.0   0   13.0   0    15.0]
```

```
>>> print(L)
```

```
[ 6.0   0     0     0     0     0     0   ]
[ 7.0  15.0   0     0     0     0     0   ]
[ 0    13.0  11.0   0     0     0     0   ]
[ 0     0     0     4.0   0     0     0   ]
[ 0     0     9.0   0     8.0   0     0   ]
[ 0     0    12.0   5.0  10.0  14.0   0   ]
[ 0     0     0     2.0   3.0   0.0   1.0]
```

Notice that L is a reordered lower-triangular representation of A. We can convert L to an `spmatrix` using the *spmatrix()* method:

```
>>> print(L.spmatrix(reordered = False))
```

```
[ 1.0   0     0     0     0     0     0   ]
[ 2.0   4.0   0     0     0     0     0   ]
[ 0     0     6.0   0     0     0     0   ]
[ 3.0   0     0     8.0   0     0     0   ]
[ 0     0     0     9.0  11.0   0     0   ]
[ 0.0   5.0   0    10.0  12.0  14.0   0   ]
[ 0     0     7.0   0    13.0   0    15.0]
```

This returns an `spmatrix` with the same ordering as A, i.e., the inverse permutation is applied to L.

The following example illustrates how to use the Cholesky routine:

```
from cvxopt import spmatrix, amd, normal
from chompack import symbolic, cspmatrix, cholesky

# generate sparse matrix and compute symbolic factorization
I = [0, 1, 3, 1, 5, 2, 6, 3, 4, 5, 4, 5, 6, 5, 6]
J = [0, 0, 0, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 6]
A = spmatrix([0.1*(i+1) for i in range(15)], I, J, (7,7)) + spmatrix(10.0,range(7),range(7))
symb = symbolic(A, p=amd.order)

# create cspmatrix
L = cspmatrix(symb)
L += A

# compute numeric factorization
cholesky(L)
```

```
>>> print(L)
```

```
[ 3.3   0     0     0     0     0     0   ]
[ 0.2   3.4   0     0     0     0     0   ]
[ 0     0.4   3.3   0     0     0     0   ]
[ 0     0     0     3.2   0     0     0   ]
[ 0     0     0.3   0     3.3   0     0   ]
[ 0     0     0.4   0.2   0.3   3.3   0   ]
[ 0     0     0     0.1   0.1  -0.0   3.2]
```

Given a sparse matrix A, we can check if it is chordal by checking whether the permutation p returned by maximum cardinality search is a perfect elimination ordering:

```
from cvxopt import spmatrix, printing
printing.options['width'] = -1
import chompack as cp

# Define chordal sparse matrix
I = range(17)+[2,2,3,3,4,14,4,14,8,14,15,8,15,7,8,14,8,14,14,\
    15,10,12,13,16,12,13,16,12,13,15,16,13,15,16,15,16,15,16,16]
J = range(17)+[0,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,\
    8,9,9,9,9,10,10,10,11,11,11,11,12,12,12,13,13,14,14,15]
A = spmatrix(1.0,I,J,(17,17))

# Compute maximum cardinality search
p = cp.maxcardsearch(A)
```

Is p a perfect elimination ordering?

```
>>> cp.peo(A,p)
```

```
True
```

Let's verify that no fill is generated by the symbolic factorization:

```
>>> symb = cp.symbolic(A,p)
>>> print(symb.fill)
```

```
(0, 0)
```

3.2 Symbolic factorization

class chompack.**symbolic**(*A*, *p=None*, *merge_function=None*, ***kwargs*)

Symbolic factorization object.

Computes symbolic factorization of a square sparse matrix A and creates a symbolic factorization object.

Parameters

- **A** – spmatrix
- **p** – permutation vector or ordering routine (optional)
- **merge_function** – routine that implements a merge heuristic (optional)

The optional argument *p* can be either a permutation vector or an ordering rutine that takes an spmatrix and returns a permutation vector.

The optional argument *merge_function* allows the user to merge supernodes in the elimination tree in a greedy manner; the argument must be a routine that takes the following four arguments and returns either *True* or *False*:

Parameters

- **cp** – clique order of the parent of clique k
- **ck** – clique order of clique k
- **np** – supernode order of the parent of supernode k
- **nk** – supernode order of supernode k

The clique k is merged with its parent if the return value is *True*.

Nsn
Number of supernodes

blkptr
Pointer array for block storage of chordal sparse matrix.

chidx
Integer array with indices of child vertices in etree: *chidx[chptr[k]:chptr[k+1]]* are the indices of the children of supernode k.

chptr
Pointer array associated with *chidx*: *chidx[chptr[k]:chptr[k+1]]* are the indices of the children of supernode k.

clique_number
The clique number (the order of the largest clique)

cliques(*reordered=True*)
Returns a list of cliques

fill
Tuple with number of lower-triangular fill edges: *fill[0]* is the fill due to symbolic factorization, and *fill[1]* is the fill due to supernodal amalgamation

ip
Inverse permutation vector

n
Number of nodes (matrix order)

nnz
Returns the number of lower-triangular nonzeros.

p
Permutation vector

parent()
Returns a supernodal parent list: the i'th element is equal to -1 if supernode i is a root node in the clique forest, and otherwise the i'th element is the index of the parent of supernode i.

relidx
The relative index array facilitates fast "extend-add" and "extract" operations in the supernodal-multifrontal algorithms. The relative indices associated with supernode k is a list of indices I such that the frontal matrix F associated with the parent of node k can be updated as *F[I,I] += Uj*. The relative indices are stored in an integer array *relidx* with an associated pointer array *relptr*.

relptr
Pointer array assoicated with *relidx*.

separators(*reordered=True*)
Returns a list of separator sets

sncolptr
Pointer array associated with *snrowidx*.

snode
Supernode array: supernode k consists of nodes *snode[snptr[k]:snptr[k+1]]* where *snptr* is the supernode pointer array

snpar
Supernode parent array: supernode k is a root of the supernodal elimination tree if *snpar[k]* is equal to k, and otherwise *snpar[k]* is the index of the parent of supernode k in the supernodal elimination tree

snpost
Supernode post-ordering

snptr
Supernode pointer array: supernode k is of order *snpptr[k+1]-snptr[k]* and supernode k consists of nodes *snode[snptr[k]:snptr[k+1]]*

snrowidx
Row indices associated with representative vertices: *snrowidx[sncolptr[k]:sncolptr[k+1]]* are the row indices in the column corresponding the the representative vertex of supernode k, or equivalently, *snrowidx[sncolptr[k]:sncolptr[k+1]]* is the k'th clique.

sparsity_pattern(*reordered=True*, *symmetric=True*)
Returns a sparse matrix with the filled pattern. By default, the routine uses the reordered pattern, and the inverse permutation is applied if *reordered* is *False*.

Parameters

- **reordered** – boolean (default: *True*)
- **symmetric** – boolean (default: *True*)

supernodes(*reordered=True*)
Returns a list of supernode sets

3.3 Chordal sparse matrices

class chompack.**cspmatrix**(*symb*, *blkval=None*, *factor=False*)
Chordal sparse matrix object.

Parameters

- **symb** – *symbolic* object
- **blkval** – matrix with numerical values (optional)
- **factor** – boolean (default is *False*)

A *cspmatrix* object contains a reference to a symbolic factorization as well as an array with numerical values which are stored in a compressed block storage format which is a block variant of the compressed column storage format.

A = cspmatrix(symb) creates a new chordal sparse matrix object with a sparsity pattern defined by the symbolic factorization object *symb*. If the optional argument *blkval* specified, the *cspmatrix* object will use the *blkval* array for numerical values (and not a copy!), and otherwise the *cspmatrix* object is initialized with an all-zero array. The optional input *factor* determines whether or not the *cspmatrix* stores a factored matrix.

add_projection(*A*, *alpha=1.0*, *beta=1.0*, *reordered=False*)
Add projection of a dense matrix A to *cspmatrix*.

X := alpha*proj(A) + beta*X

copy()
Returns a new *cspmatrix* object with a reference to the same symbolic factorization, but with a copy of the array that stores the numerical values.

diag(*reordered=True*)
Returns a vector with the diagonal elements of the matrix.

is_factor
This property is equal to *True* if the cspmatrix represents a Cholesky factor, and otherwise it is equal to *False*.

spmatrix(*reordered=True*, *symmetric=False*)
Converts the *cspmatrix* A to a sparse matrix. A reordered matrix is returned if the optional argument *reordered* is *True* (default), and otherwise the inverse permutation is applied. Only the default options are allowed if the *cspmatrix* A represents a Cholesky factor.

Parameters

- **reordered** – boolean (default: True)
- **symmetric** – boolean (default: False)

3.4 Numerical computations

chompack.**cholesky**(X)
Supernodal multifrontal Cholesky factorization:

$$X = LL^T$$

where L is lower-triangular. On exit, the argument X contains the Cholesky factor L.

Parameters **X** – *cspmatrix*

chompack.**llt**(L)
Supernodal multifrontal Cholesky product:

$$X = LL^T$$

where L is lower-triangular. On exit, the argument L contains the product X.

Parameters **L** – *cspmatrix* (factor)

chompack.**projected_inverse**(L)
Supernodal multifrontal projected inverse. The routine computes the projected inverse

$$Y = P(L^{-T}L^{-1})$$

where L is a Cholesky factor. On exit, the argument L contains the projected inverse Y.

Parameters **L** – *cspmatrix* (factor)

chompack.**completion**(X, *factored_updates=True*)
Supernodal multifrontal maximum determinant positive definite matrix completion. The routine computes the Cholesky factor L of the inverse of the maximum determinant positive definite matrix completion of X:, i.e.,

$$P(S^{-1}) = X$$

where $S = LL^T$. On exit, the argument X contains the lower-triangular Cholesky factor L.

The optional argument *factored_updates* can be used to enable (if True) or disable (if False) updating of intermediate factorizations.

Parameters

- **X** – *cspmatrix*
- **factored_updates** – boolean

chompack.**psdcompletion**(*A*, *reordered=True*, ***kwargs*)

Maximum determinant positive semidefinite matrix completion. The routine takes a cspmatrix A and returns the maximum determinant positive semidefinite matrix completion X as a dense matrix, i.e.,

$$P(X) = A$$

Parameters

- **A** – *cspmatrix*
- **reordered** – boolean

chompack.**mrcompletion**(*A*, *reordered=True*)

Minimum rank positive semidefinite completion. The routine takes a positive semidefinite cspmatrix A and returns a dense matrix Y with r columns that satisfies

$$P(YY^T) = A$$

where

$$r = \max_i |\gamma_i|$$

is the clique number (the size of the largest clique).

Parameters

- **A** – *cspmatrix*
- **reordered** – boolean

chompack.**edmcompletion**(*A*, *reordered=True*, ***kwargs*)

Euclidean distance matrix completion. The routine takes an EDM-completable cspmatrix A and returns a dense EDM X that satisfies

$$P(X) = A$$

Parameters

- **A** – *cspmatrix*
- **reordered** – boolean

chompack.**hessian**(*L*, *Y*, *U*, *adj=False*, *inv=False*, *factored_updates=False*)

Supernodal multifrontal Hessian mapping.

The mapping

$$\mathcal{H}_X(U) = P(X^{-1}UX^{-1})$$

is the Hessian of the log-det barrier at a positive definite chordal matrix X, applied to a symmetric chordal matrix U. The Hessian operator can be factored as

$$\mathcal{H}_X(U) = \mathcal{G}_X^{adj}(\mathcal{G}_X(U))$$

where the mappings on the right-hand side are adjoint mappings that map chordal symmetric matrices to chordal symmetric matrices.

This routine evaluates the mapping G_X and its adjoint G_X^{adj} as well as the corresponding inverse mappings. The inputs *adj* and *inv* control the action as follows:

Action	*adj*	*inv*
$U = \mathcal{G}_X(U)$	False	False
$U = \mathcal{G}_X^{adj}(U)$	True	False
$U = \mathcal{G}_X^{-1}(U)$	False	True
$U = (G_X^{adj})^{-1}(U)$	True	True

The input argument L is the Cholesky factor of X. The input argument Y is the projected inverse of X. The input argument U is either a chordal matrix (a *cspmatrix*) of a list of chordal matrices with the same sparsity pattern as L and Y.

The optional argument *factored_updates* can be used to enable (if True) or disable (if False) updating of intermediate factorizations.

Parameters

- **L** – *cspmatrix* (factor)
- **Y** – *cspmatrix*
- **U** – *cspmatrix* or list of *cspmatrix* objects
- **adj** – boolean
- **inv** – boolean
- **factored_updates** – boolean

chompack.**dot**(*X*, *Y*)
Computes trace product of X and Y.

chompack.**trmm**(*L*, *B*, *alpha=1.0*, *trans='N'*, *nrhs=None*, *offsetB=0*, *ldB=None*)
Multiplication with sparse triangular matrix. Computes

$$B := \alpha LB \text{ if trans is 'N'}$$
$$B := \alpha L^T B \text{ if trans is 'T'}$$

where L is a *cspmatrix* factor.

Parameters

- **L** – *cspmatrix* factor
- **B** – matrix
- **alpha** – float (default: 1.0)
- **trans** – 'N' or 'T' (default: 'N')
- **nrhs** – number of right-hand sides (default: number of columns in B)
- **offsetB** – integer (default: 0)
- **ldB** – leading dimension of B (default: number of rows in B)

chompack.**trsm**(*L*, *B*, *alpha=1.0*, *trans='N'*, *nrhs=None*, *offsetB=0*, *ldB=None*)
Solves a triangular system of equations with multiple right-hand sides. Computes

$$B := \alpha L^{-1} B \text{ if trans is 'N'}$$
$$B := \alpha L^{-T} B \text{ if trans is 'T'}$$

where L is a *cspmatrix* factor.

Parameters

- **L** – *cspmatrix* factor
- **B** – matrix
- **alpha** – float (default: 1.0)
- **trans** – 'N' or 'T' (default: 'N')
- **nrhs** – number of right-hand sides (default: number of columns in B)

- **offsetB** – integer (default: 0)
- **ldB** – leading dimension of B (default: number of rows in B)

class chompack.**pfcholesky**(*X*, *V*, *a=None*, *p=None*)

Supernodal multifrontal product-form Cholesky factorization:

$$X + V\mathrm{diag}(a)V^T = L_m \cdots L_1 L_0 L_0^T L_1^T \cdots L_m^T$$

where $X = L_0 L_0^T$ is of order n and V is n-by-m.

Parameters

- **X** – *cspmatrix* or spmatrix
- **V** – n-by-m matrix
- **a** – m-by-1 matrix (optional, default is vector of ones)
- **p** – n-by-1 matrix (optional, default is natural ordering)

trmm(*B*, *trans='N'*)

Multiplication with product-form Cholesky factor. Computes

$$B := LB \text{ if trans is 'N'}$$
$$B := L^T B \text{ if trans is 'T'}$$

trsm(*B*, *trans='N'*)

Solves a triangular system of equations with multiple righthand sides. Computes

$$B := L^{-1} B \text{ if trans is 'N'}$$
$$B := L^{-T} B \text{ if trans is 'T'}$$

3.5 Chordal conversion

The following example illustrates how to apply the chordal conversion technique to a sparse SDP.

```
# Given: tuple with cone LP problem data
# prob = (c,G,h,dims,A,b)

# Solve cone LP with CVXOPT's conelp() routine
sol = cvxopt.conelp(*prob)

# Apply chordal conversion to cone LP
probc, blk2sparse, symbs = chompack.convert_conelp(*prob)

# Solve converted problem with CVXOPT's conelp() routine
solc = cvxopt.solvers.conelp(*probc)
```

chompack.**convert_conelp**(*c*, *G*, *h*, *dims*, *A=None*, *b=None*, ***kwargs*)

Applies the clique conversion method of Fukuda et al. to the positive semidefinite blocks of a cone LP.

Parameters

- **c** – matrix
- **G** – spmatrix
- **h** – matrix

- **dims** – dictionary
- **A** – `spmatrix` or `matrix`
- **b** – `matrix`

The following example illustrates how to convert a cone LP:

```
prob = (c,G,h,dims,A,b)
probc, blk2sparse, symbs = convert_conelp(*prob)
```

The return value *blk2sparse* is a list of 4-tuples (*blki,I,J,n*) that each defines a mapping between the sparse matrix representation and the converted block-diagonal representation, and *symbs* is a list of symbolic factorizations corresponding to each of the semidefinite blocks in the original cone LP.

See also:

M. Fukuda, M. Kojima, K. Murota, and K. Nakata, Exploiting Sparsity in Semidefinite Programming via Matrix Completion I: General Framework, SIAM Journal on Optimization, 11:3, 2001, pp. 647-674.

S. Kim, M. Kojima, M. Mevissen, and M. Yamashita, Exploiting Sparsity in Linear and Nonlinear Matrix Inequalities via Positive Semidefinite Matrix Completion, Mathematical Programming, 129:1, 2011, pp.. 33-68.

`chompack.`**`convert_block`**(*G*, *h*, *dim*, ***kwargs*)

Applies the clique conversion method to a single positive semidefinite block of a cone linear program

$$\begin{array}{ll} \text{maximize} & -h^T z \\ \text{subject to} & G^T z + c = 0 \\ & \mathbf{smat}(z) \;\; \text{psd completable} \end{array}$$

After conversion, the above problem is converted to a block-diagonal one

$$\begin{array}{ll} \text{maximize} & -h_b^T z_b \\ \text{subject to} & G_b^T z_b + c = 0 \\ & G_c^T z_b = 0 \\ & \mathbf{smat}(z_b) \;\; \text{psd block-diagonal} \end{array}$$

where z_b is a vector representation of a block-diagonal matrix. The constraint $G_b^T z_b + c = 0$ corresponds to the original constraint $G'z + c = 0$, and the constraint $G_c^T z_b = 0$ is a coupling constraint.

Parameters

- **G** – `spmatrix`
- **h** – `matrix`
- **dim** – integer
- **merge_function** – routine that implements a merge heuristic (optional)
- **coupling** – mode of conversion (optional)
- **max_density** – float (default: 0.4)

The following example illustrates how to apply the conversion method to a one-block SDP:

```
block = (G, h, dim)
blockc, blk2sparse, symb = convert_block(*block)
```

The return value *blk2sparse* is a 4-tuple (*blki,I,J,n*) that defines a mapping between the sparse matrix representation and the converted block-diagonal representation. If *blkvec* represents a block-diagonal matrix, then

```
S = spmatrix(blkvec[blki], I, J)
```

maps *blkvec* into is a sparse matrix representation of the matrix. Similarly, a sparse matrix *S* can be converted to the block-diagonal matrix representation using the code

```
blkvec = matrix(0.0, (len(S),1), tc=S.typecode)
blkvec[blki] = S.V
```

The optional argument *max_density* controls whether or not to perform conversion based on the aggregate sparsity of the block. Specifically, conversion is performed whenever the number of lower triangular nonzeros in the aggregate sparsity pattern is less than or equal to *max_density*dim*.

The optional argument *coupling* controls the introduction of equality constraints in the conversion. Possible values are *full* (default), *sparse*, *sparse+tri*, and any nonnegative integer. Full coupling results in a conversion in which all coupling constraints are kept, and hence the converted problem is equivalent to the original problem. Sparse coupling yeilds a conversion in which only the coupling constraints corresponding to nonzero entries in the aggregate sparsity pattern are kept, and sparse-plus-tridiagonal (*sparse+tri*) yeilds a conversion with tridiagonal coupling in addition to coupling constraints corresponding to nonzero entries in the aggregate sparsity pattern. Setting *coupling* to a nonnegative integer *k* yields a conversion with coupling constraints corresponding to entries in a band with half-bandwidth *k*.

See also:

M. S. Andersen, A. Hansson, and L. Vandenberghe, Reduced-Complexity Semidefinite Relaxations of Optimal Power Flow Problems, IEEE Transactions on Power Systems, 2014.

3.6 Auxiliary routines

chompack.**maxcardsearch**(*A*, *ve=None*)

Maximum cardinality search ordering of a sparse chordal matrix.

Returns the maximum cardinality search ordering of a symmetric chordal matrix A. Only the lower triangular part of A is accessed. The maximum cardinality search ordering is a perfect elimination ordering in the factorization $PAP^T = LL^T$. The optional argument *ve* is the index of the last vertex to be eliminated (the default value is n-1).

Parameters

- **A** – spmatrix
- **ve** – integer between 0 and *A.size[0]*-1 (optional)

chompack.**peo**(*A*, *p*)

Checks whether an ordering is a perfect elmimination order.

Returns *True* if the permutation p is a perfect elimination order for a Cholesky factorization $PAP^T = LL^T$. Only the lower triangular part of A is accessed.

Parameters

- **A** – spmatrix
- **p** – matrix or list of length *A.size[0]*

chompack.**maxchord**(*A*, *ve=None*)

Maximal chordal subgraph of sparsity graph.

Returns a lower triangular sparse matrix which is the projection of A on a maximal chordal subgraph and a perfect elimination order p. Only the lower triangular part of A is accessed. The optional argument *ve* is the index of the last vertex to be eliminated (the default value is *n-1*). If A is chordal, then the matrix returned is equal to A.

Parameters

- **A** – spmatrix
- **ve** – integer between 0 and *A.size[0]*-1 (optional)

See also:

P. M. Dearing, D. R. Shier, D. D. Warner, Maximal chordal subgraphs, Discrete Applied Mathematics, 20:3, 1988, pp. 181-190.

chompack.**merge_size_fill**(*tsize=8*, *tfill=8*)
Simple heuristic for supernodal amalgamation (clique merging).

Returns a function that returns *True* if either (i) supernode k and supernode par(k) are both of size at most *tsize*, or (ii), merging supernodes par[k] and k induces at most *tfill* nonzeros in the lower triangular part of the sparsity pattern.

Parameters

- **tsize** – nonnegative integer; threshold for merging based on supernode sizes
- **tfill** – nonnegative integer; threshold for merging based on induced fill

chompack.**tril**(*A*)
Returns the lower triangular part of A.

chompack.**triu**(*A*)
Returns the upper triangular part of A.

chompack.**perm**(*A*, *p*)
Symmetric permutation of a symmetric sparse matrix.

Parameters

- **A** – spmatrix
- **p** – matrix or list of length *A.size[0]*

chompack.**symmetrize**(*A*)
Returns a symmetric matrix from a sparse square matrix A. Only the lower triangular entries of A are accessed.

CHAPTER 4

Examples

4.1 SDP conversion

This example demonstrates the SDP conversion method. We first generate a random sparse SDP:

```
from cvxopt import matrix, spmatrix, sparse, normal, solvers, blas
import chompack as cp
import random

# Function for generating random sparse matrix
def sp_rand(m,n,a):
    """
    Generates an m-by-n sparse 'd' matrix with round(a*m*n) nonzeros.
    """
    if m == 0 or n == 0: return spmatrix([], [], [], (m,n))
    nnz = min(max(0, int(round(a*m*n))), m*n)
    nz = matrix(random.sample(range(m*n), nnz), tc='i')
    return spmatrix(normal(nnz,1), nz%m, nz/m, (m,n))

# Generate random sparsity pattern and sparse SDP problem data
random.seed(1)
m, n = 50, 200
A = sp_rand(n,n,0.015) + spmatrix(1.0,range(n),range(n))
I = cp.tril(A)[:].I
N = len(I)/50 # each data matrix has 1/50 of total nonzeros in pattern
Ig = []; Jg = []
for j in range(m):
    Ig += sorted(random.sample(I,N))
    Jg += N*[j]
G = spmatrix(normal(len(Ig),1),Ig,Jg,(n**2,m))
h = G*normal(m,1) + spmatrix(1.0,range(n),range(n))[:]
c = normal(m,1)
dims =  {'l':0, 'q':[], 's': [n]};
```

The problem can be solved using CVXOPT's cone LP solver:

```
prob = (c, G, matrix(h), dims)
sol = solvers.conelp(*prob)
Z1 = matrix(sol['z'], (n,n))
```

```
     pcost       dcost       gap    pres   dres   k/t
 0: -7.9591e+00 -2.9044e+02  3e+02  6e-16  2e+00  1e+00
 1: -1.1826e+01 -1.1054e+02  9e+01  4e-15  8e-01  4e-01
```

```
 2: -1.7892e+01 -9.9108e+01  8e+01  4e-15  7e-01  3e-01
 3: -2.3947e+01 -4.1603e+01  2e+01  2e-15  1e-01  8e-02
 4: -2.4646e+01 -3.7190e+01  1e+01  3e-15  1e-01  6e-02
 5: -2.6001e+01 -2.9393e+01  3e+00  3e-15  3e-02  2e-02
 6: -2.6000e+01 -2.9051e+01  3e+00  3e-15  3e-02  1e-02
 7: -2.6252e+01 -2.6796e+01  6e-01  2e-15  5e-03  3e-03
 8: -2.6272e+01 -2.6506e+01  2e-01  3e-15  2e-03  1e-03
 9: -2.6288e+01 -2.6345e+01  6e-02  2e-15  5e-04  3e-04
10: -2.6289e+01 -2.6326e+01  4e-02  3e-15  3e-04  2e-04
11: -2.6291e+01 -2.6296e+01  5e-03  3e-15  4e-05  3e-05
12: -2.6291e+01 -2.6292e+01  1e-03  3e-15  9e-06  6e-06
13: -2.6291e+01 -2.6291e+01  1e-04  2e-15  9e-07  6e-07
14: -2.6291e+01 -2.6291e+01  9e-06  2e-15  8e-08  5e-08
Optimal solution found.
```

An alternative is to convert the sparse SDP into a block-diagonal SDP using the conversion method and solve the converted problem using CVXOPT:

```
prob2, blocks_to_sparse, symbs = cp.convert_conelp(*prob)
sol2 = solvers.conelp(*prob2)
```

```
     pcost       dcost       gap    pres   dres   k/t
 0: -7.9953e+00 -2.8985e+02  1e+03  9e-01  2e+00  1e+00
 1: -1.8054e+01 -8.9659e+01  1e+02  2e-01  6e-01  1e+00
 2: -2.2574e+01 -5.4864e+01  5e+01  1e-01  3e-01  7e-01
 3: -2.4720e+01 -3.4637e+01  1e+01  3e-02  8e-02  2e-01
 4: -2.5730e+01 -2.8667e+01  4e+00  9e-03  2e-02  4e-02
 5: -2.6107e+01 -2.6927e+01  1e+00  3e-03  7e-03  9e-03
 6: -2.6252e+01 -2.6400e+01  2e-01  5e-04  1e-03  1e-03
 7: -2.6279e+01 -2.6322e+01  5e-02  1e-04  4e-04  3e-04
 8: -2.6289e+01 -2.6295e+01  7e-03  2e-05  5e-05  3e-05
 9: -2.6291e+01 -2.6293e+01  2e-03  6e-06  2e-05  7e-06
10: -2.6291e+01 -2.6292e+01  5e-04  1e-06  3e-06  1e-06
11: -2.6291e+01 -2.6291e+01  1e-04  2e-07  6e-07  2e-07
12: -2.6291e+01 -2.6291e+01  2e-05  6e-08  2e-07  6e-08
13: -2.6291e+01 -2.6291e+01  5e-06  1e-08  3e-08  1e-08
Optimal solution found.
```

The solution to the original SDP can be found by mapping the block-diagonal solution to a sparse positive semidefinite completable matrix and computing a positive semidefinite completion:

```
# Map block-diagonal solution sol2['z'] to a sparse positive semidefinite completable matrix
blki,I,J,bn = blocks_to_sparse[0]
Z2 = spmatrix(sol2['z'][blki],I,J)

# Compute completion
symb = cp.symbolic(Z2, p=cp.maxcardsearch)
Z2c = cp.psdcompletion(cp.cspmatrix(symb)+Z2, reordered=False)
Y2 = cp.mrcompletion(cp.cspmatrix(symb)+Z2, reordered=False)
```

The conversion can also be combined with clique-merging techniques in the symbolic factorization. This typically yields a block-diagonal SDP with fewer (but bigger) blocks than without clique-merging:

```
mf = cp.merge_size_fill(5,5)
prob3, blocks_to_sparse, symbs = cp.convert_conelp(*prob, coupling = 'full', merge_function = mf)
sol3 = solvers.conelp(*prob3)
```

```
     pcost       dcost       gap    pres   dres   k/t
 0: -8.2758e+00 -2.9164e+02  6e+02  6e-01  2e+00  1e+00
 1: -1.8634e+01 -8.0009e+01  8e+01  1e-01  5e-01  9e-01
 2: -2.4495e+01 -4.5428e+01  3e+01  4e-02  2e-01  3e-01
 3: -2.5371e+01 -3.2361e+01  8e+00  1e-02  6e-02  5e-02
 4: -2.5903e+01 -2.8828e+01  3e+00  6e-03  2e-02  2e-02
 5: -2.6226e+01 -2.6782e+01  6e-01  1e-03  5e-03  3e-03
 6: -2.6275e+01 -2.6391e+01  1e-01  2e-04  1e-03  6e-04
 7: -2.6289e+01 -2.6308e+01  2e-02  4e-05  2e-04  1e-04
 8: -2.6291e+01 -2.6294e+01  3e-03  6e-06  2e-05  1e-05
 9: -2.6291e+01 -2.6292e+01  1e-03  2e-06  7e-06  4e-06
10: -2.6291e+01 -2.6291e+01  1e-04  2e-07  7e-07  4e-07
11: -2.6291e+01 -2.6291e+01  9e-06  2e-08  7e-08  4e-08
Optimal solution found.
```

Finally, we recover the solution to the original SDP:

```
# Map block-diagonal solution sol2['z'] to a sparse positive semidefinite completable matrix
blki,I,J,bn = blocks_to_sparse[0]
Z3 = spmatrix(sol3['z'][blki],I,J)

# Compute completion
symb = cp.symbolic(Z3, p=cp.maxcardsearch)
Z3c = cp.psdcompletion(cp.cspmatrix(symb)+Z3, reordered=False)
```

4.2 Euclidean distance matrix completion

Suppose that A is a partial EDM of order n where the squared distance $A_{ij} = \|p_i - p_j\|_2^2$ between two point p_i and p_j is known if p_i and p_j are sufficiently close. We will assume that A_{ij} is known if and only if

$$\|p_i - p_j\|_2^2 \leq \delta$$

where δ is a positive constant. Let us generate a random partial EDM based on points in $\mathbb{R}^2$:

```
from cvxopt import uniform, spmatrix, matrix
import chompack as cp

d = 2               # dimension
n = 100             # number of points (order of A)
delta = 0.15**2     # distance threshold

P = uniform(d,n)    # generate n points with independent and uniformly distributed coordinates
Y = P.T*P           # Gram matrix

# Compute true distances:  At[i,j] = norm(P[:,i]-P[:,j])**2
#   At = diag(Y)*ones(1,n) + ones(n,1)*diag(Y).T - 2*Y
At = Y[::n+1]*matrix(1.0,(1,n)) + matrix(1.0,(n,1))*Y[::n+1].T - 2*Y

# Generate matrix with "observable distances"
#   A[i,j] = At[i,j] if At[i,j] <= delta
V,I,J = zip(*[(At[i,j],i,j) for j in range(n) for i in range(j,n) if At[i,j] <= delta])
A = spmatrix(V,I,J,(n,n))
```

The partial EDM A may or may not be chordal. We can find a maximal chordal subgraph using the `maxchord` routine which returns a chordal matrix A_c and a perfect elimination order p. Note that if A is chordal, then $A_c = A$.

```
Ac,p = cp.maxchord(A)
```

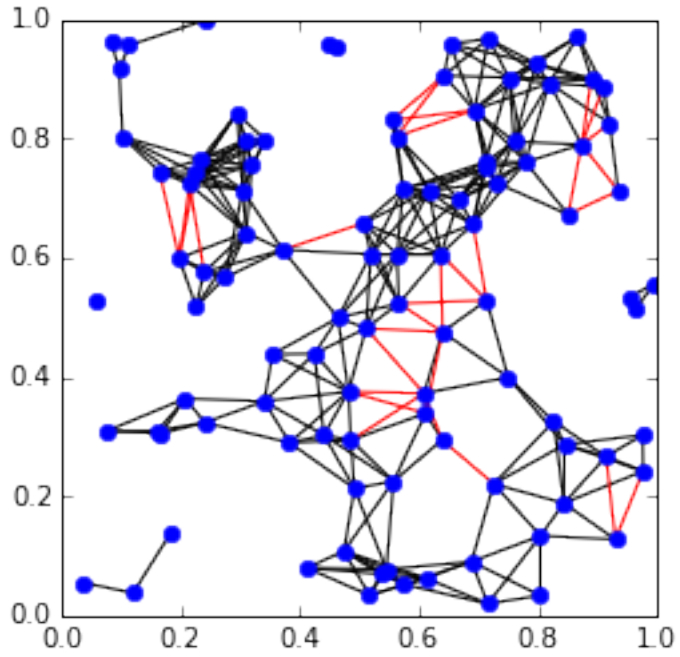
The points p_i and the known distances can be visualized using Matplotlib:

```
from pylab import plot,xlim,ylim,gca

# Extract entries in Ac and entries dropped from A
IJc = zip(Ac.I,Ac.J)
tmp = A - Ac
IJd = [(i,j) for i,j,v in zip(tmp.I,tmp.J,tmp.V) if v > 0]

# Plot edges
for i,j in IJc:
    if i > j: plot([P[0,i],P[0,j]],[P[1,i],P[1,j]],'k-')
for i,j in IJd:
    if i > j: plot([P[0,i],P[0,j]],[P[1,i],P[1,j]],'r-')

# Plot points
plot(P[0,:].T,P[1,:].T, 'b.', ms=12)
xlim([0.,1.])
ylim([0.,1.])
gca().set_aspect('equal')
```



The edges represent known distances. The red edges are edges that were removed to produce the maximal chordal subgraph, and the black edges are the edges of the chordal subgraph.

Next we compute a symbolic factorization of the chordal matrix A_c using the perfect elimination order p:

```
symb = cp.symbolic(Ac, p=p)
p = symb.p
```

Now `edmcompletion` can be used to compute an EDM completion of the chordal matrix A_c:

```
X = cp.edmcompletion(cp.cspmatrix(symb)+Ac, reordered = False)
```

4.3 Symbolic factorization

This example demonstrates the symbolic factorization. We start by generating a test problem and computing a symbolic factorization using the approximate minimum degree (AMD) ordering heuristic:

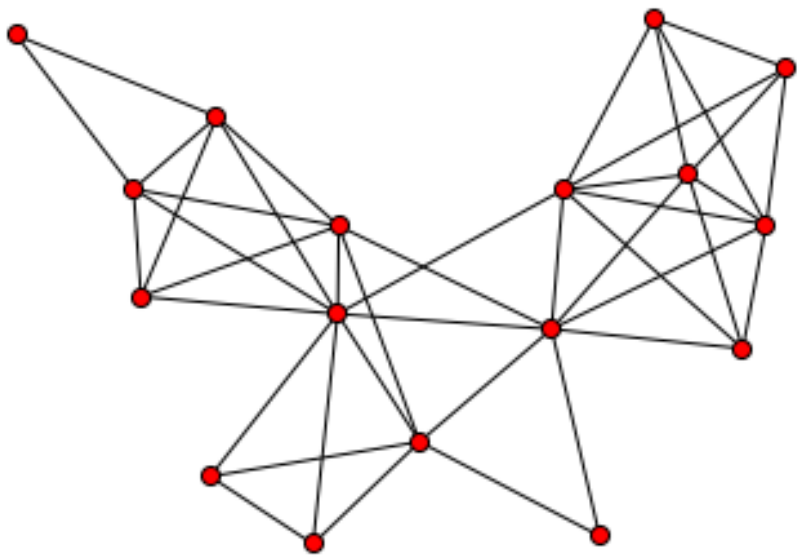
```
import chompack as cp
from cvxopt import spmatrix, amd

L = [[0,2,3,4,14],[1,2,3],[2,3,4,14],[3,4,14],[4,8,14,15],[5,8,15],[6,7,8,14],[7,8,14],[8,14,15],[9,1
I = []
J = []
for k,l in enumerate(L):
    I.extend(l)
    J.extend(len(l)*[k])

A = spmatrix(1.0,I,J,(17,17))
symb = cp.symbolic(A, p=amd.order)
```

The sparsity graph can be visualized with the `sparsity_graph` routine if Matplotlib, NetworkX, and Graphviz are installed:

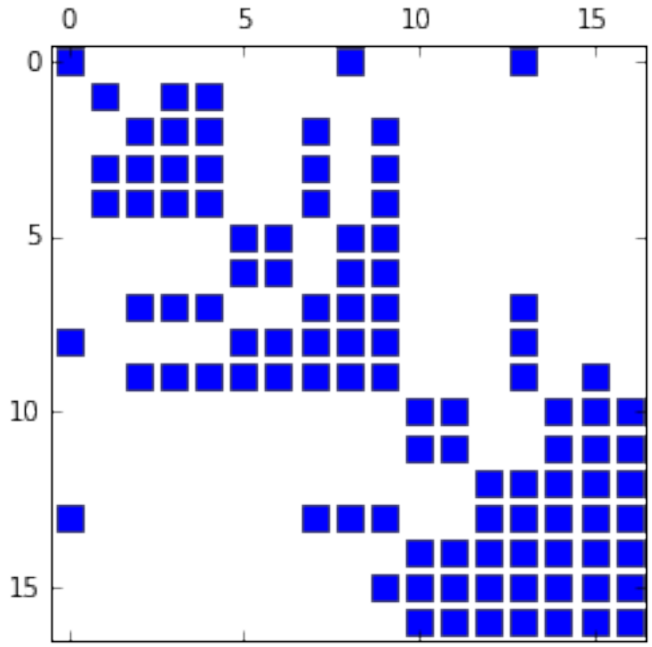
```
from chompack.pybase.plot import sparsity_graph
sparsity_graph(symb, node_size=50, with_labels=False)
```



The `sparsity_graph` routine passes all optional keyword arguments to NetworkX to make it easy to customize the visualization.

It is also possible to visualize the sparsity pattern using the `spy` routine which requires the packages Matplotlib, Numpy, and Scipy:

```
from chompack.pybase.plot import spy
fig = spy(symb, reordered=True)
```



The supernodes and the supernodal elimination tree can be extracted from the symbolic factorization as follows:

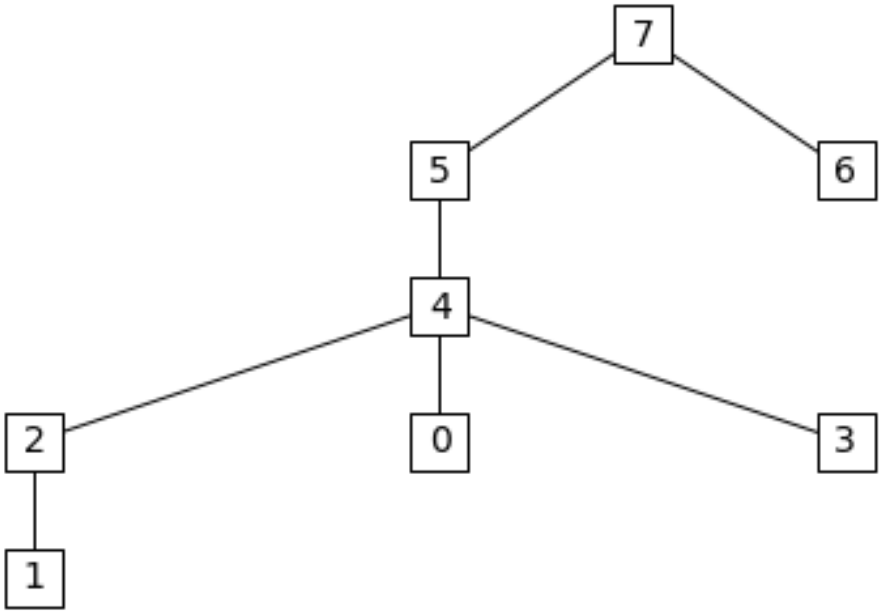
```
par = symb.parent()
snodes = symb.supernodes()

print "Id  Parent id  Supernode"
for k,sk in enumerate(snodes):
    print "%2i      %2i      "%(k,par[k]), sk
```

```
Id  Parent id  Supernode
 0       4       [0]
 1       2       [1]
 2       4       [2, 3, 4]
 3       4       [5, 6]
 4       5       [7, 8]
 5       7       [9]
 6       7       [10, 11]
 7       7       [12, 13, 14, 15, 16]
```

The supernodal elimination tree can be visualized with the `etree_graph` routine if Matplotlib, NetworkX, and Graphviz are installed:

```
from chompack.pybase.plot import etree_graph
etree_graph(symb, with_labels=True, arrows=False, node_size=500, node_color='w', node_shape='s', font
```


7
5
6
4
2
0
3
1

C