SMCP Documentation

Release 0.4.5

Martin S. Andersen and Lieven Vandenberghe

May 18, 2018

Contents

1 Current release **3**

2 Future releases **5**

3 Availability **7**

4 Authors **9**

5 Feedback and bug reports **11**

5.1 Copyright and license 11

5.2 Download and installation 11

5.3 Documentation 12

5.4 Test problems 20

5.5 Benchmarks 20

SMCP is a software package for solving linear sparse matrix cone programs. The code is experimental and it is released to accompany the following paper:

See also:

13. S. Andersen, J. Dahl, and L. Vandenberghe, Implementation of nonsymmetric interior-point methods for linear optimization over sparse matrix cones, *Mathematical Programming Computation*, 2010.

The package provides an implementation of a nonsymmetric interior-point method which is based on chordal matrix techniques. Only one type of cone is used, but this cone includes the three canonical cones — the nonnegative orthant, the second-order cone, and the positive semidefinite cone — as special cases. The efficiency of the solver depends not only on the dimension of the cone, but also on its *structure*. Nonchordal sparsity patterns are handled using chordal embedding techniques.

In its current form, SMCP is implemented in Python and C, and it relies on the Python extensions CHOMPACK and CVXOPT for most computations.

CHAPTER 1

Current release

Version 0.4.5 (December 2017) includes:

- Nonsymmetric feasible start interior-point methods (primal and dual scaling methods)
- Two KKT system solvers: one solves the symmetric indefinite augmented system and the other solves the positive definite system of normal equations
- Read/write routines for SDPA sparse data files ('dat-s').
- Simple interface to CVXOPT SDP solver

CHAPTER 2

Future releases

We plan to turn SMCP into a C library with Python and Matlab interfaces. Future releases may include additional functionality as listed below:

- Explicitly handle free variables
- Iterative solver for the KKT system
- Automatic selection of KKT solver and chordal embedding technique

CHAPTER 3

Availability

The source package is available from the *Download and installation* section. The source package includes source code, documentation, and installation guidelines.

CHAPTER 4

Authors

SMCP is developed by Martin S. Andersen and Lieven Vandenberghe .

CHAPTER 5

Feedback and bug reports

We welcome feedback, and bug reports are much appreciated. Please report bugs through our Github repository.

5.1 Copyright and license

Copyright 2009-2018 M. Andersen and L. Vandenberghe.

SMCP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

SMCP is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

5.2 Download and installation

5.2.1 Installation with pip

SMCP can be installed via pip using the following command:

```
pip install smcp
```

5.2.2 Installation from source

The package requires Python version 2.7 or newer. To build the package from source, the Python header files and libraries must be installed, as well as the core binaries. SMCP also requires the Python extension modules CVXOPT 1.1.9 or later and CHOMPACK 2.3.2 or later.

The source package is available here. The extension can be built and installed as follows:

```
python setup.py install
python example.py
```

5.3 Documentation

5.3.1 Overview

Let $\mathbf{S}^n$ denote the set of symmetric matrices of order n, and let and let $C \bullet X$ denote the standard inner product on $\mathbf{S}^n$.

$\mathbf{S}^n_V$ is the set of symmetric matrices of order n and with sparsity pattern V, *i.e.*, $X \in \mathbf{S}^n_V$ if and only if $X_{ij} = 0$ for all $(i,j) \neq V$. We will assume that all diagonal elements are included in V. The projection Y of X on the subspace $\mathbf{S}^n_V$ is denoted $Y = P_V(X)$, *i.e.*, $Y_{ij} = X_{ij}$ if $(i,j) \in V$ and $Y_{ij} = 0$ otherwise. The inequality signs $\succeq$ and $\succ$ denote matrix inequality. We define $|V|$ as the number of nonzeros in the lower triangular part of V, and $\mathrm{nnz}(A_i)$ denotes the number of nonzeros in the matrix A_i.

$\mathbf{S}^n_{V,+}$ and $\mathbf{S}^n_{V,++}$ are the sets of positive semidefinite and positive definite matrices in $\mathbf{S}^n_V$, and similarly, $\mathbf{S}^n_{V,c+} = \{P_V(X) \mid X \succeq 0\}$ and $\mathbf{S}^n_{V,c++} = \{P_V(X) \mid X \succ 0\}$ are the sets of matrices in $\mathbf{S}^n_V$ that have a positive semidefinite completion and a positive definite completion, respectively. We denote with $\succeq_{\mathrm{c}}$ and $\succ_{\mathrm{c}}$ matrix inequality with respect to the cone $\mathbf{S}^n_{V,c+}$.

SMCP solves a pair of primal and dual linear cone programs:

$$\begin{array}{lll} (P) & \text{minimize} & C \bullet X \\ & \text{subject to} & A_i \bullet X = b_i, i = 1, \ldots, m \\ & & X \succeq_{\mathrm{c}} 0 \end{array} \qquad \begin{array}{lll} (D) & \text{maximize} & b^T y \\ & \text{subject to} & \sum_{i=1}^m y_i A_i + S = C \\ & & S \succeq 0. \end{array}$$

The variables are $X \in \mathbf{S}^n_V$, $S \in \mathbf{S}^n_V$, and $y \in \mathbf{R}^m$, and the problem data are the matrix $C \in \mathbf{S}^n_V$, the vector $b \in \mathbf{R}^m$, and $A_i \in \mathbf{S}^n_V, i = 1, \ldots, m$.

Compositions of cones are handled implicitly by defining a block diagonal sparsity pattern V. Dense blocks and general sparse blocks correspond to standard positive semidefinite matrix constraints, diagonal blocks corresponds to linear inequality constraints, and second-order cone constraints can be embedded in an LMI with an "arrow pattern", *i.e*,

$$\|x\|_2 \leq t \quad \Leftrightarrow \quad \begin{bmatrix} t & x^T \\ x & tI \end{bmatrix} \succeq 0.$$

5.3.2 The chordal SDP solvers

`smcp.solvers.`**`chordalsolver_feas`**(*A*, *b*[, *primalstart*[, *dualstart*[, *scaling='primal'*[, *kktsolver='chol'*]]]])

Solves the pair of cone programs (P) and (D) using a feasible start interior-point method. If no primal and/or dual feasible starting point is specified, the algorithm tries to find a feasible starting point based on some simple heuristics. An exception is raised if no starting point can be found. In this case a Phase I problem must be solved, or the (experimental) infeasible start interior-point method `chordalsolver_esd` can be used.

The columns of the sparse matrix `A` are vectors of length n^2 and the $m+1$ columns of `A` are:

$$[\mathbf{vec}(C)\ \mathbf{vec}(A_1)\ \cdots\ \mathbf{vec}(A_m)]\,.$$

Only the rows of `A` corresponding to the lower triangular elements of the aggregate sparsity pattern V are accessed.

The optional argument *primalstart* is a dictionary with the key *x* which can be used to specify an initial value for the primal variable X. Similarly, the optional argument *dualstart* must be a dictionary with keys *s* and *y*.

The optional argument *scaling* takes one of the values `'primal'` (default) or `'dual'`.

The optional argument *kktsolver* is used to specify the KKT solver. Possible values include:

`'chol'` (default) solves the KKT system via a Cholesky factorization of the Schur complement

`'qr'` solves the KKT system via a QR factorization

The solver returns a dictionary with the following keys:

`'primal objective'`, `'dual objective'` primal objective value and dual objective value.

`'primal infeasibility'`, `'dual infeasibility'` residual norms of primal and dual infeasibility.

`'x'`, `'s'`, and `'y'` primal and dual variables.

`'iterations'` number of iterations.

`'cputime'`, `'time'` total cputime and real time.

`'gap'` duality gap.

`'relative gap'` relative duality gap.

`'status'`

- has the value `'optimal'` if

$$\frac{\|b-\mathcal{A}(X)\|_2}{\max\{1,\|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|\mathcal{A}^{\text{adj}}(y)+S-C\|_F}{\max\{1,\|C\|_F\}} \leq \epsilon_{\text{feas}}, \qquad X \succ_{\text{c}} 0, \qquad S \succ 0,$$

  and

$$X \bullet S \leq \epsilon_{\text{abs}} \quad \text{or} \quad \left(\min\{C\bullet X, -b^T y\} \leq 0, \frac{X\bullet S}{-\min\{C\bullet X, -b^T y\}} \leq \epsilon_{\text{rel}}\right).$$

- has the value `'unknown'` otherwise.

The following options can be set using the dictionary `smcp.solvers.options`:

`'delta'` (default: 0.9) a positive constant between 0 and 1; an approximate tangent direction is computed when the Newton decrement is less than `delta`.

`'eta'` (default: `None`) `None` or a positive float. If `'eta'` is a positive number, a step in the approximate tangent direction is taken such that

$$\Omega(X+\alpha\Delta X, S+\alpha\Delta S) \approx \eta$$

where $\Omega(X,S)$ is the proximity function

$$\Omega(X,S) = \phi_{\text{c}}(X) + \phi(S) + n\cdot\log\frac{X\bullet S}{n} + n.$$

If `'eta'` is `None`, the step length α in the approximate tangent direction is computed as

$$\begin{aligned}\alpha_p &= \arg\max\{\alpha\in(0,1] \,|\, X+\alpha\Delta X \succeq_{\text{c}} 0\}\\ \alpha_d &= \arg\max\{\alpha\in(0,1] \,|\, S+\alpha\Delta S \succeq 0\}\\ \alpha &= \texttt{step}\cdot\min(\alpha_p,\alpha_d)\end{aligned}$$

where `step` is the value of the option `'step'` (default: 0.98).

`'prediction'` (default: `True`) `True` or `False`. This option is effective only when `'eta'` is `None`. If `'prediction'` is `True`, a step in the approximate tangent direction is never taken but only used to predict the duality gap. If `'prediction'` is `False`, a step in the approximate tangent direction is taken.

`'step'` (default: 0.98) positive float between 0 and 1.

`'lifting'` (default: `True`) `True` or `False`; determines whether or not to apply lifting before taking a step in the approximate tangent direction.

`'show_progress'` `True` or `False`; turns the output to the screen on or off (default: `True`).

`'maxiters'` maximum number of iterations (default: `100`).

`'abstol'` absolute accuracy (default: `1e-6`).

`'reltol'` relative accuracy (default: `1e-6`).

`'feastol'` tolerance for feasibility conditions (default: `1e-8`).

`'refinement'` number of iterative refinement steps when solving KKT equations (default: `1`).

`'cholmod'` use Cholmod's AMD embedding (defaults: `False`).

`'dimacs'` report DIMACS error measures (default: `True`).

`smcp.solvers.`**`chordalsolver_esd`**(*A*, *b*[, *primalstart*[, *dualstart*[, *scaling='primal'*[, *kktsolver='chol'*]]]])

Solves the pair of cone programs (P) and (D) using an extended self-dual embedding. This solver is currently experimental.

The columns of the sparse matrix `A` are vectors of length n^2 and the $m+1$ columns of `A` are:

$$[\mathbf{vec}(C)\ \mathbf{vec}(A_1)\ \cdots\ \mathbf{vec}(A_m)]\,.$$

Only the rows of `A` corresponding to the lower triangular elements of the aggregate sparsity pattern V are accessed.

The optional argument *primalstart* is a dictionary with the key *x* which can be used to specify an initial value for the primal variable X. Similarly, the optional argument *dualstart* must be a dictionary with keys *s* and *y*.

The optional argument *scaling* takes one of the values `'primal'` (default) or `'dual'`.

The optional argument *kktsolver* is used to specify the KKT solver. Possible values include:

`'chol'` (default) solves the KKT system via a Cholesky factorization of the Schur complement

`'qr'` solves the KKT system via a QR factorization

The solver returns a dictionary with the following keys:

`'primal objective'`, `'dual objective'` primal objective value and dual objective value.

`'primal infeasibility'`, `'dual infeasibility'` residual norms of primal and dual infeasibility.

`'x'`, `'s'`, and `'y'` primal and dual variables.

`'iterations'` number of iterations.

`'cputime'`, `'time'` total cputime and real time.

`'gap'` duality gap.

`'relative gap'` relative duality gap.

'status'

- has the value `'optimal'` if

$$\frac{(1/\tau)\|\tau b - \mathcal{A}(X)\|_2}{\max\{1, \|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{(1/\tau)\|\mathcal{A}^{\text{adj}}(y) + S - \tau C\|_F}{\max\{1, \|C\|_F\}} \leq \epsilon_{\text{feas}}, \qquad X \succ_{\text{c}} 0, \qquad S \succ 0,$$

  and

$$\frac{X \bullet S}{\tau^2} \leq \epsilon_{\text{abs}} \quad \text{or} \quad \left(\min\{C \bullet X, -b^T y\} \leq 0, \frac{(1/\tau) X \bullet S}{-\min\{C \bullet X, -b^T y\}} \leq \epsilon_{\text{rel}}\right).$$

- has the value `'primal infeasible'` if

$$b^T y = 1, \qquad \frac{\|\mathcal{A}^{\text{adj}}(y) + S\|_F}{\max\{1, \|C\|_F\}} \leq \epsilon_{\text{feas}}, \qquad S \succ 0.$$

- has the value `'dual infeasible'` if

$$C \bullet X = -1, \qquad \frac{\|\mathcal{A}(X)\|_2}{\max\{1, \|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad X \succ_{\text{c}} 0.$$

- has the value `'unknown'` if maximum number iterations is reached or if a numerical error is encountered.

The following options can be set using the dictionary `smcp.solvers.options`:

'show_progress' `True` or `False`; turns the output to the screen on or off (default: `True`).

'maxiters' maximum number of iterations (default: `100`).

'abstol' absolute accuracy (default: `1e-6`).

'reltol' relative accuracy (default: `1e-6`).

'feastol' tolerance for feasibility conditions (default: `1e-8`).

'refinement' number of iterative refinement steps when solving KKT equations (default: `1`).

'cholmod' use Cholmod's AMD embedding (defaults: `False`).

'dimacs' report DIMACS error measures (default: `True`).

5.3.3 Solver interfaces

The following functions implement CVXOPT-like interfaces to the experimental solver *chordalsolver_esd*.

`smcp.solvers.`**`conelp`**(*c*, *G*, *h*[, *dims*[, *kktsolver='chol'*]])
Interface to *chordalsolver_esd*.

`smcp.solvers.`**`lp`**(*c*, *G*, *h*[, *kktsolver='chol'*])
Interface to *conelp*; see CVXOPT documentation for more information.

`smcp.solvers.`**`socp`**(*c*[, *Gl*, *hl*[, *Gq*, *hq*[, *kktsolver='chol'*]]])
Interface to *conelp*; see CVXOPT documentation for more information.

`smcp.solvers.`**`sdp`**(*c*[, *Gl*, *hl*[, *Gs*, *hs*[, *kktsolver='chol'*]]])
Interface to *conelp*; see CVXOPT documentation for more information.

5.3.4 The SDP object

class SDP (*filename*)

Class for SDP problems. Simplifies reading and writing SDP data files and includes a wrapper for `chordalsolver_esd`.

The constructor accepts sparse SDPA data files (extension 'dat-s') and data files created with the `save` method (extension 'pkl'). Data files compressed with Bzip2 can also be read (extensions 'dat-s.bz2' and 'pkl.bz2').

m

Number of constraints.

n

Order of semidefinite variable.

A

Problem data: sparse matrix of size $n^2 \times (m+1)$ with columns $\mathbf{vec}(C), \mathbf{vec}(A_1), \ldots, \mathbf{vec}(A_m)$. Only the lower triangular elements of $C, A_1, \ldots, A_m$ are stored.

b

Problems data: vector of length m.

V

Sparse matrix with aggregate sparsity pattern (lower triangle).

nnz

Number of nonzero elements in lower triangle of aggregate sparsity pattern.

nnzs

Vector with number of nonzero elements in lower triangle of $A_0, \ldots, A_m$.

nzcols

Vector with number of nonzero columns in $A_1, \ldots, A_m$.

issparse

True if the number of nonzeros is less than $0.5 \cdot n(n+1)/2$, otherwise false.

ischordal

True if aggregate sparsity pattern is chordal, otherwise false.

get_A (i)

Returns the i'th coeffiecient matrix A_i ($0 \leq i \leq m$) as a sparse matrix. Only lower triangular elements are stored.

write_sdpa ([*fname*[, *compress=False*]])

Writes SDP data to SDPA sparse data file. The extension 'dat-s' is automatically added to the filename. The method is an interface to `sdpa_write`.

If `compress` is true, the data file is compressed with Bzip2 and 'bz2' is appended to the filename.

save ([*fname*[, *compress=False*]])

Writes SDP data to file using cPickle. The extension 'pkl' is automatically added to the filename.

If `compress` is true, the data file is compressed with Bzip2 and 'bz2' is appended to the filename.

solve_feas ([*scaling='primal'*[, *kktsolver='chol'*[, *primalstart*[, *dualstart*]]]])

Interface to the feasible start solver `chordalsolver_feas`. Returns dictionary with solution.

solve_phase1 ([*kktsolver='chol'*[, *M=1e5*]])

Solves a Phase I problem to find a feasible (primal) starting point:

$$\begin{array}{ll} \text{minimize} & s \\ \text{subject to} & A_i \bullet X = b_i, \quad i = i, \ldots, m \\ & \mathbf{tr}(X) \leq M \\ & X + (s - \epsilon) I \succeq_{\mathrm{c}} 0, \; s \geq 0 \end{array}$$

The variables are $X \in \mathbf{S}_V^n$ and $s \in \mathbf{R}$, and $\epsilon \in \mathbf{R}_{++}$ is a small constant. If $s^\star < \epsilon$, the method returns $X^\star$ (which is a strictly feasible starting point in the original problem) and a dictionary (with information about the Phase I problem). If $s >= \epsilon$ the method returns (*None*, *None*).

solve_esd([*scaling='primal'*[, *kktsolver='chol'*[, *primalstart*[, *dualstart*]]]])

Interface to `chordalsolver_esd`. Returns dictionary with solution.

solve_cvxopt([*primalstart*[, *dualstart*]])

Interface to `cvxopt.solvers.sdp()`. Returns dictionary with solution. (Note that this simple interface does not yet specify block structure properly.)

The following example demostrates how to load and solve a problem from an SDPA sparse data file:

```
>>> from smcp import SDP
>>> P = SDP('qpG11.dat-s')
>>> print P
<SDP: n=1600, m=800, nnz=3200> qpG11
>>> sol = P.solve_feas(kktsolver='chol')
>>> print sol['primal objective']
-2448.6588977
>>> print sol['dual objective']
-2448.65913565
>>> print sol['gap']
0.00023794772363
>>> print sol['relative gap']
9.71747121876e-08
```

5.3.5 Auxiliary routines

smcp.**completion**(*X*)

Computes the maximum determinant positive definite completion of a sparse matrix X.

Example:

```
>>> from smcp import mtxnorm_SDP, completion
>>> P = mtxnorm_SDP(p=10,q=2,r=10)
>>> sol = P.solve_feas(kktsolver='chol')
>>> X = completion(sol['x'])
```

smcp.misc.**ind2sub**(*siz*, *ind*)

Converts indices to subscripts.

Parameters

- **siz** (*integer*) – matrix order
- **ind** (*matrix*) – vector with indices

Returns matrix `I` with row subscripts and matrix `J` with column subscripts

smcp.misc.**sub2ind**(*siz*, *I*, *J*)

Converts subscripts to indices.

Parameters

- **siz** (*integer tuple*) – matrix size
- **I** (*matrix*) – row subscripts
- **J** (*matrix*) – column subscripts

Returns matrix with indices

smcp.misc.**sdpa_read**(*file_obj*)

Reads data from sparse SDPA data file (file extension: 'dat-s'). A description of the sparse SDPA file format can be found in the document SDPLIB/FORMAT and in the SDPA User's Manual.

Example:

```
>>> f = open('qpG11.dat-s')
>>> A, b, blockstruct = smcp.misc.sdpa_read(f)
>>> f.close()
```

smcp.misc.**sdpa_readhead**(*file_obj*)

Reads header from sparse SDPA data file and returns the order n, the number of constraints m, and a vector with block sizes.

Example:

```
>>> f = open('qpG11.dat-s')
>>> n, m, blockstruct = smcp.misc.sdpa_readhead(f)
>>> f.close()
```

smcp.misc.**sdpa_write**(*file_obj*, *A*, *b*, *blockstruct*)

Writes SDP data to sparse SDPA file.

Example:

```
>>> f = open('my_data_file.dat-s','w')
>>> smcp.misc.sdpa_write(f,A,b,blockstruct)
>>> f.close()
```

5.3.6 Analysis routines

smcp.analysis.**embed_SDP**(*P*[, *order*[, *cholmod*]])

Computes chordal embedding and returns SDP object with chordal sparsity pattern.

Parameters

- **P** (*SDP*) – SDP object with problem data
- **order** (*string*) – 'AMD' (default) or 'METIS'
- **cholmod** (*boolean*) – use Cholmod to compute embedding (default is false)

Returns SDP object with chordal sparsity

Note that CVXOPT must be compiled and linked to METIS in order to use the METIS ordering.

The following routines require Matplotlib:

smcp.analysis.**spy**(*P*[, *i*[, *file*[, *scale*]]])

Plots aggregate sparsity pattern of SDP object `P` or sparsity pattern of A_i.

Parameters

- **P** (*SDP*) – SDP object with problem data
- **i** (*integer*) – index between 0 and m
- **file** (*string*) – saves plot to file
- **scale** (*float*) – downsamples plot

smcp.analysis.**clique_hist**(*P*)

Plots clique histogram if `P.ischordal` is true, and otherwise an exception is raised.

Parameters **P** (*SDP*) – SDP object with problem data

smcp.analysis.**nnz_hist**(*P*)

Plots histogram of number of nonzeros in lower triangle of $A_1, \ldots, A_m$.

Parameters **P** (*SDP*) – SDP object with problem data

5.3.7 Random problem generators

class mtxnorm_SDP(*p*, *q*, *r*[, *density*[, *seed*]])

Inherits from *SDP* class.

Generates random data $F_i, G \in \mathbf{R}^{p \times q}$ for the matrix norm minimization problem

$$\text{minimize} \quad \|F(z) + G\|_2$$

with the variable $z \in \mathbf{R}^r$ and where $F(z) = z_1 F_1 + \cdots z_r F_r$. The problem is cast as an equivalent SDP:

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & \begin{bmatrix} tI & (F(z)+G)^T \\ F(z)+G & tI \end{bmatrix} \succeq 0. \end{array}$$

The sparsity of F_i can optionally be chosen by specifying the parameter `density` which must be a float between 0 and 1 (default is 1 which corresponds to dense matrices).

Example:

```
>>> from smcp import mtxnorm_SDP
>>> P = mtxnorm_SDP(p=200,q=10,r=200)
>>> print P
<SDP: n=210, m=201, nnz=2210> mtxnorm_p200_q10_r200
>>> sol = P.solve_feas(kktsolver='qr')
```

class base.**band_SDP**(*n*, *m*, *bw*[, *seed*])

Generates random SDP with band sparsity and *m* constraints, of order *n*, and with bandwidth *bw* (*bw=0* corresponds to a diagonal, *bw=1* is tridiagonal etc.). Returns *SDP* object. The optional parameter *seed* sets the random number generator seed.

Example:

```
>>> from smcp import band_SDP
>>> P = band_SDP(n=100,m=100,bw=2,seed=10)
>>> print P
<SDP: n=100, m=100, nnz=297> band_n100_m100_bw2
>>> X,p1sol = P.solve_phase1(kktsolver='qr')
>>> P.solve_feas(kktsolver='qr',primalstart={'x':X})
>>> print sol['primal objective'],sol['dual objective']
31.2212701455 31.2212398351
```

class `base.`**`rand_SDP`**(*V*, *m*[, *density*[, *seed*]])

Generates random SDP with sparsity pattern V and m constraints. Returns `SDP` object.

The sparsity of A_i can optionally be chosen by specifying the parameter `density` which must be a float between 0 and 1 (default is 1 which corresponds to dense matrices).

5.4 Test problems

The SMCP repository contains a number of SDP problem instances that were created with SMCP and have been used for *benchmarks*. The files follow the SDPA sparse data format and are compressed with Bzip2.

5.5 Benchmarks

To assess the performance of our preliminary implementation of SMCP, we have conducted a series of numerical experiments.

5.5.1 SDP solvers

The following interior-point solvers were used in our experiments:

- Method **M1** (SMCP 0.3a, feasible start solver with *kktsolver='chol'*)
- Method **M1c** (SMCP 0.3a, feasible start solver with *kktsolver='chol'* and *solvers.options['cholmod']=True*)
- Method **M2** (SMCP 0.3a, feasible start solver with *kktsolver='qr'*)
- CSDP 6.0.1
- DSDP 5.8
- SDPA 7.3.1
- SDPA-C 6.2.1 (binary dist.)
- SDPT3 4.0b (64-bit Matlab)
- SeDuMi 1.2 (64-bit Matlab)

5.5.2 Error measures

We report DIMACS error measures when available. The six error measures are defined as:

$$
\begin{aligned}
\epsilon_1(X,y,S) &= \frac{\|\mathcal{A}(X)-b\|_2}{1+\|b\|_\infty} \\
\epsilon_2(X,y,S) &= \max\left\{0, \frac{-\lambda_{\min}(X)}{1+\|b\|_\infty}\right\} \\
\epsilon_3(X,y,S) &= \frac{\|\mathcal{A}^{\mathrm{adj}}(y)+S-C\|_F}{1+\|C\|_{\max}} \\
\epsilon_4(X,y,S) &= \max\left\{0, \frac{-\lambda_{\min}(S)}{1+\|C\|_{\max}}\right\} \\
\epsilon_5(X,y,S) &= \frac{C\bullet X - b^T y}{1+|C\bullet X|+|b^T y|} \\
\epsilon_6(X,y,S) &= \frac{S\bullet X}{1+|C\bullet X|+|b^T y|}
\end{aligned}
$$

Here $\|C\|_{\max} = \max_{i,j}|C_{ij}|$, and $C\bullet X = \mathbf{tr}(C^T X)$.

Note that $\epsilon_2(X,y,S)=0$ and $\epsilon_4(X,y,S)=0$ since all iterates (X,y,S) satisfy $X\in\mathbf{S}^n_{V,c++}$ and $S\in\mathbf{S}^n_{V,++}$.

5.5.3 Experimental setup

The following experiments were conducted on a desktop computer with an Intel Core 2 Quad Q6600 CPU (2.4 GHz), 4 GB of RAM, and running Ubuntu 9.10 (64 bit).

The problem instances used in the experiments are available for download *here* and the SDPLIB problems are available here.

We use the least-norm solution to the set of equations $A_i\bullet X, i=1,\ldots,m$, as starting point when it is strictly feasible, and otherwise we solve the phase I problem

$$
\begin{array}{ll}
\text{minimize} & s \\
\text{subject to} & A_i\bullet X = b_i, \quad i=1,\ldots,m, \\
& \mathbf{tr}(X)\le M \\
& X+(s-\epsilon)I \succeq_{\mathrm{c}} 0, s\ge 0.
\end{array}
$$

Here ϵ is a small constant, and the constraint $\mathbf{tr}(X)\le M$ is added to bound the feasible set.

5.5.4 SDPs with band structure

We consider a family of SDP instances where the data matrices $C, A_1,\ldots,A_m$ are of order n and banded with a common bandwidth $2w+1$.

Experiment 1

$m=100$ constraints, bandwidth 11 ($w=5$), and variable order n

Experiment 2

order $n=500$, bandwidth 7 ($w=3$), and variable number of constraints m

The problem *band_n500_m800_w3* required a phase I (M1 311.5 sec.; M2 47.8 sec.).

Experiment 3

order $n = 200$, $m = 100$ constraints, and variable bandwidth $2w + 1$

Two problems required a phase I: *band_n200_m100_w0* (M1 1.12 sec.; M2 0.53 sec.) and *band_n200_m100_w1* (M1 3.18 sec.; M2 1.45 sec.).

5.5.5 Matrix norm minimization

We consider the matrix norm minimization problem

$$\text{minimize } \|F(x) + G\|_2$$

where $F(x) = x_1 F_1 + \cdots + x_r F_r$ and $G, F_i \in \mathbf{R}^{p \times q}$ are the problem data. The problem can be formulated as an SDP:

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & \begin{bmatrix} tI & F(x) + G \\ (F(x) + G)^T & tI \end{bmatrix} \succeq 0. \end{array}$$

This SDP has dimensions $m = r + 1$ and $n = p + q$. We generate G as a dense $p \times q$ matrix, and the matrices F_i are generated such that the number of nonzero entries in each matrix is given by $\max(1, dpq)$ where the parameter $d \in [0, 1]$ determines sparsity. The locations of nonzero entries in F_i are random. Thus, the problem family is parameterized by the tuple (p, q, r, d).

Experiment 4

variable number of rows p, $q = 10$ columns, $r = 100$ variables, and density $d = 1$

Experiment 5

$p = 400$ rows, $q = 10$ columns, $r = 200$ variables, and variable density

Experiment 6:

$p = 400$ rows, $q = 10$ columns, variable number of variables r, and density $d = 1$

Experiment 7:

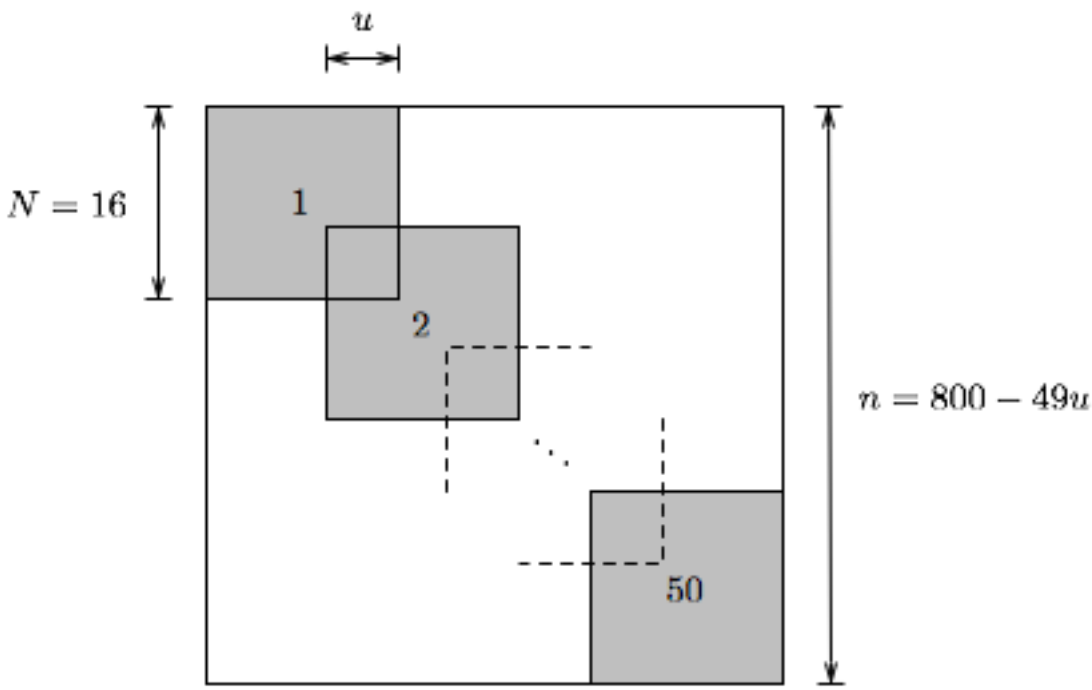
$p + q = 1000$, $r = 10$ variables, and density $d = 1$

5.5.6 Overlapping cliqes

We consider a family of SDPs which have an aggregate sparsity patterns V with l cliques of order N. The cliques are given by

$$W_i = \{(i-1)(N-u) + 1, \ldots, (i-1)(N-u) + 1 + u\}, \quad i = 1, \ldots, l$$

where u ($0 \leq u \leq N - 1$) is the overlap between neighboring cliques. The sparsity pattern is illustrated below:



Note that $u = 0$ corresponds to a block diagonal sparsity pattern and $u = N - 1$ corresponds to a band pattern.

Experiment 8

$m = 100$ constraints, clique size $N = 16$, and variable overlap u

5.5.7 SDPLIB problems

The following experiment is based on problem instances from .

Experiment 9

SDPLIB problems with $n \geq 500$

Three problems required a phase I: *thetaG11* (M1 227.2 sec.; M1c 184.4 sec.), *thetaG51* (M1 64.8 sec.: M1c 58.0 sec.), and *truss8* (M1 17.9 sec.; M1c 17.9 sec.).

5.5.8 Nonchordal sparsity patterns

The following problems are based on sparsity patterns from the University of Florida Sparse Matrix Collection (UFSMC). We use as problem identifier the name *rsX* where *X* is the ID number of a sparsity pattern from UFSMC. Each problem instance has m constraints and the number of nonzeros in the lower triangle of A_i is $\max\{1, \mathrm{round}(0.005|V|)\}$ where $|V|$ is the number of nonzeros in the lower triangle of the aggregate sparsity pattern V, and C has $|V|$ nonzeros.

Experiment 10

Index

A

A (SDP attribute), 16

B

b (SDP attribute), 16
base.band_SDP (built-in class), 19
base.rand_SDP (built-in class), 19

G

get_A() (SDP method), 16

I

ischordal (SDP attribute), 16
issparse (SDP attribute), 16

M

m (SDP attribute), 16
mtxnorm_SDP (built-in class), 19

N

n (SDP attribute), 16
nnz (SDP attribute), 16
nnzs (SDP attribute), 16
nzcols (SDP attribute), 16

S

save() (SDP method), 16
SDP (built-in class), 16
smcp.analysis.clique_hist() (built-in function), 19
smcp.analysis.embed_SDP() (built-in function), 18
smcp.analysis.nnz_hist() (built-in function), 19
smcp.analysis.spy() (built-in function), 18
smcp.completion() (built-in function), 17
smcp.misc.ind2sub() (built-in function), 17
smcp.misc.sdpa_read() (built-in function), 18
smcp.misc.sdpa_readhead() (built-in function), 18
smcp.misc.sdpa_write() (built-in function), 18
smcp.misc.sub2ind() (built-in function), 17
smcp.solvers.chordalsolver_esd() (built-in function), 14
smcp.solvers.chordalsolver_feas() (built-in function), 12
smcp.solvers.conelp() (built-in function), 15
smcp.solvers.lp() (built-in function), 15
smcp.solvers.sdp() (built-in function), 15
smcp.solvers.socp() (built-in function), 15
solve_cvxopt() (SDP method), 17
solve_esd() (SDP method), 17
solve_feas() (SDP method), 16
solve_phase1() (SDP method), 16

V

V (SDP attribute), 16

W

write_sdpa() (SDP method), 16