CBF_pythonic

Johanna Skåntorp

This is a Python library for generating a text-file in the CBF-format for Mixed-Integer Semidefinite Programs (MISDPs). The idea is to provide a simple, Pythonic, way of constructing an MISDP in the CBF-format. The contents are sorted in – what I consider – chronological order :)

Contents

1 Mathematical Formulation

The type of problems suitable can be stated as

$$\min \quad \sum_{j\in J}\langle F_j^{obj}, X_j\rangle + \sum_{\ell\in L} a_\ell^{obj} x_\ell + b^{obj} \tag{1}$$

$$\text{s.t.} \quad \sum_{j\in J}\langle F_j^{i}, X_j\rangle + \sum_{\ell\in L} a_\ell^{i} x_\ell + b^{i} \in \mathcal{K}^i, \qquad i=1,\dots,m_a, \tag{2}$$

$$\sum_{\ell\in L} H_\ell^i x_\ell + D^i \succeq 0 \qquad i=1,\dots,m_d, \tag{3}$$

$$X_j \succeq 0,\ \forall j\in J, \tag{4}$$

$$x_\ell \in \mathbb{Z},\ \forall \ell \in L_{\mathcal{I}} \subseteq L, \tag{5}$$

where

$$\begin{aligned}
\mathcal{K} &= \text{ either " } \geq 0\text{", " } = 0\text{", or " } \leq 0\text{"} \\
J &= \text{ set of PSD matrix-variables} \\
L &= \text{ set of non-matrix variables} \\
L_{\mathcal{I}} &= \text{ index set for integer variables} \\
\\
m_a &= \text{ number of affine constraints} \\
m_d &= \text{ number of PSD-constraints} \\
\\
X_j &= \text{ PSD matrix variable of size } n_j \times n_j \\
F_j &= \text{ symmetric matrix of size } n_j \times n_j \\
F_j^i &= \text{ symmetric matrix of size } n_j \times n_j \\
H_\ell^i &= \text{ symmetric matrix of size } n_i \times n_i \\
D^i &= \text{ symmetric matrix of size } n_i \times n_i \\
a^{obj},\ a^i &= \text{ vectors of length } |L| \\
b^{obj},\ b^i &= \text{ real numbers}
\end{aligned}$$

Problems can be stated in both primal and dual form, i.e., it allows for both PSD-constrained matrix variables as in (4) and linear PSD-constraints as in (3). Note that while a problem can contain both PSD-variables and PSD-constraints the (primal) PSD matrix variables defined in (4) cannot be involved in the (dual) PSD constraints of (3).

Note that apart from the PSD-constraints, all remaining terms are linear.

2 TLDR;

This is not a real optimizer, only a way to write CBF-files, so there isn't a lot of fancy stuff. This section covers most of it, and is enough to get you started

```
# from CBF_pythonic import Model
M = Model()

# addVars(n, vtype = "C", lb=-np.inf, ub=np.inf)
x = M.addvars(3, vtype = ["C", "I", "B"], lb = [0,0,0], ub = [2,2,1])

# addPSDVar(n):  also adds primal PSD-constraint
# To add 'vtype', 'lb', and 'ub' to PSD-variables, see Section 7
X1 = M.addPSDVar(4)
X2 = M.addPSDVar(2)

# Linear expressions are one-dimensional
# LinExpr:  sum <F_j, X_j> + sum a_j * x_j + b
F1 = np.eye(4) # numpy array of size (n,n)
F2 = np.ones((2,2)) # must be symmetric or lower triangular
a = np.ones(3) # numpy array of size n
lin_expr = F1*X1 + F2*X2 + a@x + 7 # NOTE: @ as vector operator

# addConstraint()
lin_con = M.addConstraint(lin_expr <= 0) # '<=', '>=', '=='

# addPSDConstraint():  dual PSD-constraint
H0 = np.eye(3) # numpy array (symmetric or lower triangular)
H1 = np.ones((3,3)) # all matrices in one constraint must be same size
D = np.array([[1,3],[3,2]])
mat_expr = D + H0*x[0] + H1*x[1]
psd_con = M.addPSDConstraint(mat_expr)

# addObjective(sense, LinExpr)
lin_expr = np.array([7,3]) @ x[[0,4]] # you can slice variables
objective = M.addObjective("MIN", lin_expr) # 'MIN' or 'MAX'

M.writeCBF(file_name) # Write model to file
```

Some important notes:

- All matrices must be $n \times n$ numpy arrays. **They must be symmetric or lower triangular!**
- All matrices H_i, D in the same constraint i must be the same size;
- All vectors a must be numpy arrays;
- Vector multiplication is defined using the `@`-operator;
- **Linear expressions are 1-D:** there is no way to input $Ax = b$;
- Adding a PSD-variable X also adds the constraint $X \succeq 0$

You cannot query the `Model` to get variables or constraints, so you have to keep track of them yourself. What you can do is print out constraints etc., to see what they look like in the CBF-file:

```
# Use .print to print cbf-type output
Model.print # prints the cbf-file so far
# Constraints and expressions can also be printed:
LinConstraint.print, PSDConstraint.print, LinExpr.print
```

3 The `Model` Class

```
from CBF_pythonic import Model
M = Model()
```

Public methods

```
# Main methods
M.addVar(vtype, lb, ub) # Add single variable                              (4)
M.addVars(n, vtype, lb, ub) # Add n variables                              (4)
M.addPSDvar(n) # Add nxn matrix variable                                   (7)
M.addConstraint(LinConstraint) # Add linear constraint                    (10)
M.addPSDConstraint(MatExpr) # Add PSD-constraint                          (15)
M.addObjective(objectivesense, LinExpr) # Add objective                   (16)

M.writeCBF(file_name) # Write model to file                               (16)

# Other
M.copy() # Copy model
M.print # prints out the cbf-file so far
M.output_string() # returns the string from writeCBF()
```

Other classes:

```
SingleVar # single variable                                                (5)
Variable # vector variable                                                 (5)
PSDVar # primal nxn matrix variable                                        (8)
LinExpr # linear expression                                                (9)
LinConstraint # linear constraint                                         (11)
MatExpr # matrix expression                                               (12)
PSDConstraint # PSD dual constraint                                       (14)
```

4 Adding Variables: `addVars()`, `addVar()`

Non-matrix variables are added using the functions

```
M.addVars(n, vtype="C", lb=-np.inf, ub=np.inf) # returns Variable if n > 1,
                                               # else returns SingleVar
M.addVar(vtype="C", lb=-np.inf, ub=np.inf) # returns SingleVar
```

For $n = 1$ `addVars()` and `addVar()` are equivalent.

- `n` (variable size):
  - Format: `int`
- `vtype` (variable type):
  - Format: `str` or `list of strs` of length `n`
  - Options: `"C"` (continuous), `"B"` (binary), and `"I"` (integer)
  - Default: `"C"`
- `lb` (lower bound):
  - Format: `float` or `list of floats` of length `n`
  - Default: `"-np.inf"`
- `ub` (upper bound):
  - Format: `float` or `list of floats` of length `n`
  - Default: `"np.inf"`

In the CBF-format there is no way to denote binary variables, so they are reassigned as integer (`"I"`) and the bounds are adjusted accordingly, i.e.,

```
if vtype == "B":
     vtype = "I"
     lb = max(lb, 0)
     ub = min(ub, 1)
```

Examples

The variable $x = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$, where

$$x_1 \in \{0, 1\}, \text{ and}$$
$$0 \leq x \leq 3$$

can be added either as a vector

```
# add variables as vector
x = M.addVars(3, ["B", "C", "C"], 0, 3)
```

or as individual variables

```
# add individual variables
x1 = M.addVar("B", lb=0, ub=3) # ub = 1 is chosen automatically
x2 = M.addVar("C", lb=0, ub=3)
x3 = M.addVar("C", lb=0, ub=3)
```

Similarly we can add

$$y \in \mathbb{Z}^2$$
$$0 \leq y_1 \leq 3$$
$$y_2 \leq 5$$

either as a vector

```
# add variables as vector
y = M.addVars(2, vtype = "I", lb=[0, -np.inf], ub=[3,5])
```

or as individual variables

```
# add individual variables
y1 = M.addVars(1, vtype = "I", lb=0, ub=3)
y2 = M.addVars(1, vtype = "I", ub=5)
```

5 The `Variable` Class

```
# Public methods
Variable.size # returns variable-size 'n'
Variable.index # returns list of assigned indices
```

Index assignment

Indexing of variables is automatic and done (in order) at creation

```
# Example of index assignment
x = M.addVars(3)
print(x.index) # out:  [0,1,2]
y = M.addVars(2)
print(y.index) # out:  [3,4]
```

Supported operations

Slicing

You can slice a `Variable` instance the same way you could a numpy array, using `__getitem__()`. This returns a new `Variable` instance (or `SingleVar` if $n = 1$)

```
# Examples of slicing
x = M.addVars(5)
print(x.index) # out:  [0,1,2,3,4]

y = x[0:3]
print(y.index) # out:  [0,1,2]

z = x[[1,3,4]]
print(z.index) # out:  [1,3,4]
print(type(z)) # out:  Variable

w = x[-1]
print(w.index) # out:  4
print(type(w)) # out:  SingleVar
```

Vector Multiplication

Instances of the `Variable` class (of size n) can be multiplied with a numpy array of dimension $(n,)$ using `__matmul__()` or `__rmatmul__()`, i.e., the `@`-operator

```
# Vector multiplication:  '@'
x = M.addVars(5)
a = np.ones(5)
lin_expr = a @ x # returns a linear expression:  LinExpr
```

`LinExpr` are explained in Section 9.

6 The `SingleVar` Class

```
# Public methods
SingleVar.size # returns '1'
SingleVar.index # returns assigned index
```

Index assignment

Indexing of variables is automatic and done (in order) at creation

```
# Example of index assignment
x = M.addVars(3)
print(x.index) # out:  [0,1,2]
y = M.addVar()
print(y.index) # out:  3
```

Supported operations

Matrix Multiplication

All linear constraints in the model are one-dimensional, which means there is no matrix-vector multiplication supported. Instances of the `SingleVar` class can be multiplied with a matrix using `__mul__()` or `__rmul__()`, i.e., the `*`-operator

```
# Matrix multiplication:  '*'
x = M.addVars(1)
H = np.ones((5,5))
mat_expr = H * x # returns a matrix expression:  MatExpr
```

For implementation tips regarding the `Variable` instance, see examples in Section 12, where the `MatExpr` class is explained.

Multiplication, Addition, Comparison

Additionally all operations supported by `LinExpr` are also supported by `SingleVar`, see Section 9.

Developers note

The `SingleVar` and `Variable` classes are basically just indices. They only exist in order to support operations (such as addition, multiplication, etc). Importantly the `Model` does not have access to them – so you have to keep track of them!

7 Adding PSD-variables: `addPSDVar()`

PSD-constrained matrix variables $X \in \mathcal{S}_+^n$ are added by calling

```
# add n x n matrix variable constrained to the PSD cone
M.addPSDVar(n) # returns PSDVar
```

Note that adding a variable

```
X = M.addPSDVar(n)
```

adds the constraint $X \succeq 0$, as well.

Integer PSD-variables

Note that the PSD-variables cannot be directly constrained to be integer. To constraint $X_{ij} \in \mathbb{Z}$ we would have to add additional variable $y \in \mathbb{Z}$ and add the affine constraint

$$\langle X, E_{ij} \rangle = y \tag{6}$$

where

$$E_{ij} = \frac{1}{2}(e_i e_j^\top + e_j e_i^\top) \tag{7}$$

Upper and lower bound

Direct assignment of upper and lower bound for PSD-variables has not been implemented. Let LB be a matrix s.t.

$$LB_{ij} \leq X_{ij} \tag{8}$$

then we can add a lower bound as

$$\langle E_{ij}, LB \rangle, \ \leq \langle E_{ij}, X \rangle \forall (i,j).$$

Both (6) and (8) are implemented in Section 10

8 The `PSDVar` Class

```
# Public methods
PSDVar.size # returns variable-size '(n,n)'
PSDVar.index # returns assigned index PSDVar.print
```

Index assignment

Indexing of variables is automatic and done (in order) at creation. PSD-variables and single variables might share the same index, but they are different

```
# Example of index assignment
X = M.addPSDVar(3)
print(X.index) # out:  0
Y = M.addPSDVar(4)
print(Y.index) # out:  1
z = M.addVar()
print(z.index) # out:  0
```

Supported operations

Trace

The inner product (trace) between instances of the `PSDVar` class (of size $n \times n$) and numpy arrays of the same dimensions is supported. The array must either be symmetric or lower triangular. This is done using `__mul__()` or `__rmul__()`, i.e., the `*`-operator

```
# Trace:  '*'
X = M.addPSDVar(5)
A = np.ones((5,5))
lin_expr = A * X # returns a linear expression:  LinExpr
```

The `LinExpr` class is explained in Section 9.

9 The `LinExpr` Class

```
# Public methods
LinExpr.print # prints the linear expression
```

Because it is important, we again note that all linear expressions are one-dimensional. They are on the form

$$\sum_{j \in J} \langle F_j, X_j \rangle + \sum_{\ell \in L} a_\ell x_\ell + b.$$

Supported operations

These operations are also supported by the `SingleVar` class.

Addition and Subtraction

Instances of the `LinExpr` class can be added with a scalar or `LinExpr` using `__add__()` or `__radd__()`, i.e., the `+`-operator

```
# Addition '+'
lin_expr = LinExpr + 1.2 # returns LinExpr
lin_expr = LinExpr - LinExpr # returns LinExpr
lin_expr += LinExpr # returns LinExpr
```

Scalar Multiplication

Instances of the `LinExpr` class can be multiplied with a scalar using `__mul__()` or `__rmul__()`, i.e., the `*`-operator

```
# Multiplication '*'
lin_expr = 1.3 * LinExpr # returns LinExpr
```

Comparison

The comparison operators (`'<='`, `'>='`, and `'=='`) turns linear expressions into generators (`generator`), which are used to construct linear constraints, see Section 11.

Examples

Vector Examples

We give some examples of how to construct `LinExpr` using non-matrix variables, i.e.,

$$\sum_{\ell \in L} a_\ell x_\ell.$$

This is created using `numpy` arrays. Consider the linear expression

$$\begin{bmatrix} 1 & 2 & 0 \end{bmatrix} x$$

It can be constructed using the `@`-operator, which defines the dot-product

```
# Create linear expression using vector-multiplication
x = M.addVars(3)
a = np.array([1, 2, 0]) # a must be a numpy array of dim (n,)
lin_expr = a @ x
```

We can also slice the variable and use the multiplication operator `*`.

```
# Create linear expression using slice
x = M.addVars(3)
a = np.array([1, 2, 0])
lin_expr = a[0]*x[0] + a[1]*x[1] + a[2]*x[2]
```

Slicing vector variables

The slice-operator works similarly as in `numpy` meaning that, in addition to slicing with `int`s or `slice`s, we can use `list`s to slice. As an example, for $x \in \mathbb{R}^5$ the linear expression

$$x_0 + x_1 + 2x_2 + 7$$

can – apart from the previously mentioned ways:

```
x = M.addVars(5)
lin_expr = x[0] + x[1] + 2*x[2] + 7
```

and

```
a = np.array([1, 1, 2, 0, 0])
lin_expr = a @ x + 7
```

– also be constructed in the following ways:

```
a = np.array([1, 1, 2])
lin_expr = a @ x[0:3] + 7
```

and

```
lin_expr = a @ x[[0,1,2]] + 7
```

which all will have the same output:

```
lin_expr.print
# out:
> > FCOORD
> > # no F-matrices
> >
> > ACOORD
> > j val
> > 0 1.0
> > 1 1.0
> > 2 2.0
> >
> > BCOORD
> > val
> > 7.0
```

For further reading see Section 5.

Matrix Examples

We again note that if X is an $n \times n$-variable, then F must be a numpy.array of size (n,n). F must either be symmetric *or* lower-tridiagonal. Also that the $*$-operator when used with PSD-variables defines the trace operator.

We want to input

$$\sum_{j \in J} \langle F_j, X_j \rangle + b.$$

For $|J| = 1$, let

```
X = M.addPSDVar(2)
F = np.ones((2,2))
```

then we can input the expression

$$\left\langle \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, X \right\rangle + 3$$

as

```
lin_expr = F * X + 3.
```

If $|J| = 3$, given a list of matrices, we can use summation:

```
sizes = [2, 3, 3]
F = [np.eye(n) for n in sizes]
X = [M.addPSDVar(n) for n in sizes]
lin_expr = sum(F[j]*X[j] for j in range(3))
```

10 Add Linear Constraints: `addConstraint()`

Linear constraints (2) are added using

```
M.addConstraint(generator) # returns a LinConstraint
```

where a `generator` is created by comparing a linear expression (`LinExpr`) with either another `LinExpr` or a scalar. The comparison operators are:

- less than or equal to `<=`,
- equal to `==`, or
- greater than or equal to `>=`.

Again, since linear expressions containing the trace operator are one-dimensional (i.e., $\langle F, X\rangle \in \mathbb{R}$), we limit ourselves to dealing with one-dimensional linear expressions.
This has the consequence that a constraint such as

$$\begin{bmatrix}1 & 2 & 0\\ 3 & 3 & 3\end{bmatrix} x \geq \begin{bmatrix}5\\ 6\end{bmatrix}$$

must be input as two separate constraints;

$$\begin{bmatrix}1 & 2 & 0\end{bmatrix} x \geq 5$$

and

$$\begin{bmatrix}3 & 3 & 3\end{bmatrix} x \geq 6,$$

respectively.

Examples

Let

```
X = M.addPSDVar(2)
x = M.addVars(3)
F = np.ones((2,2))
a = np.array([2,3,4])
```

then we can input the constraint

$$\left\langle \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}, X \right\rangle + \begin{bmatrix}2 & 3 & 4\end{bmatrix} x == 0$$

as

```
lin_expr = F * X + a @ x
M.addConstraint(lin_expr == 0)
```

The implementation of (6) would look like

```
# E_ij is the nxn-matrix as discussed in (7)
y = M.addVars(1, "I")
X = M.addPSDVar(n)
M.addConstraint(E_ij * X == y)
```

and the implementation of (8) would look like

```
def E_matrix(i,j):
     # pseudo function that returns E_ij as in (7)

# LB is an nxn numpy array
X = M.addPSDVar(n)
for i in range(n):
     for j in range(n):
          E = E_matrix(i, j)
          M.addConstraint(E * X >= E * LB)
```

11 The `LinearConstraint` Class

A `LinearConstraint` is a glorified `LinExpr`, but you can at least print the constraint

```
# Public methods
LinConstraint.print # prints the linear constraint in cbf-style
```

The difference between `LinConstraint.print` and LinExpr.print is

- `LinConstraint.print` will include constraint-index "i", and
- `LinConstraint.print` will include "sense".

11.0.1 Developers note:

Technically a `generator` is simply a `LinConstraint` without the constraint-index

```
generator = x >= 0
type(generator)
# out:  > > type:  LinConstraint
generator.index
# out:  > > False
lin_con = M.addConstraint(generator)
lin_con == generator
# out:  > > True
```

So please don't re-use generator expressions :)

12 The `MatExpr` Class

```
# Public Methods
MatExpr.print # prints the matrix expression in cbf-style
```

Matrix expressions are on the form

$$\sum_{\ell \in L} H_\ell x_\ell + D,$$

where all H_ℓ and D matrices in the same constraint must be of the same size n. Each must be a numpy.array of size (n, n), and must be either symmetric *or* lower-tridiagonal.

They are constructed using symmetric $n \times n$ matrices, non-matrix variables and the `*`-operator. The expression

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} x_1 + \begin{bmatrix} 0 & 3 \\ 3 & 0 \end{bmatrix} x_3 - \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$$

can be written as

```
x = M.addVars(4)
H1 = np.array([[1, 2], [2, 1]])
H3 = np.array([[0, 3], [3, 0]])
D = np.array([[5, 0], [0, 5]])
mat_expr = H1 * x[1] + H3 * x[3] - D # returns MatExpr
```

Additionally, given a list of m matrices `H_list` and a matrix `D` we can construct the expression

$$\sum H_i x_i + D$$

as

```
x = M.addVars(m)
mat_expr = sum([H_list[i] * x[i] for i in range(m)]) + D # returns MatExpr
```

In addition `MatExpr` supports multiplication with a scalar

```
c = 2.3
c * MatExpr # returns MatExpr
```

13 Add PSD-constraint (dual): `addPSDConstraint()`

The PSD-constraints on dual form (3) are added using

```
M.addPSDConstraint(MatExpr) # returns PSDConstraint
```

14 The `PSDConstraint` Class

```
# Public Methods
PSDConstraint.print # prints the constraint
PSDConstraint.index # returns the index of the constraint
```

15 Add PSD-constraint (primal) (see `addPSDVar()`)

The PSD-constraints on primal form: $X \succeq 0$ are added at the same time as the variable is added, i.e., it is inherent to the PSD-variable: see 7.

16 Add Objective Function: `addObjective()`

The objective is added using

```
obj_func = M.addObjective(objsense, LinExpr) # returns Objective
```

where `objsense` is either `"MIN"` or `"MAX"` and `LinExpr` is a linear expression as above.

`Objective` Class: Public Methods

```
Objective.print # prints objective in CBF-style
```

17 Write to File: `writeCBF()`

Once the model is constructed a text-file in the CBF-format can be generated by calling

```
# Generate text-file with problem stated in CBF-format
M.writeCBF(file_name)
```

I mostly use this for trouble-shooting, but if you only want the output as a string you can use

```
M.output_string() # returns a string of the problem
```

18 Examples

Random

We solve randomly generated instances of

$$\begin{aligned} \min \quad & a^\top y + b^\top z \\ \text{s.t.} \quad & \sum_{j=1}^{m_c} H_j y_j + \sum_{j=1}^{m_b} G_j z_j + D \succeq 0, \\ & y \in [0,1]^{m_c},\ z \in \{0,1\}^{m_b}, \end{aligned}$$

where D is chosen such that the problem has at least one optimal solution.

```
def make_parameters(n, mc, mb):
    """
    Constructs randomized parameters a, b, H_j, G_j, and D
    Args:
        n (int):  size (n x n) of PSD-constraint
        mc (int):  number of continuous variables [0,1]
        mb (int):  number of binary variables 0,1

    return a, b, H, G, D
        a (ndarray):  numpy array of length mc, list elements are scalars
        b (ndarray):  numpy array of length mb, list elements are scalars
        H (list):     list of length mc,
                      list elements are matrices H_j:  (n x n) numpy array
        G (list):     list of length mb,
                      list elements are matrices G_j:  (n x n) numpy array
        D (ndarray):  (n x n) numpy array:  matrix D
    """

file_name="temp_cbf.cbf" # File name should end in .cbf
n, mc, mb = 5, 3, 3
a, b, H, G, D = make_parameters(n, mc, mb)

M = Model() # Initialize model
y = M.addVars(mc, vtype="C", lb=0, ub=1) # Add mc continuous variables
z = M.addVars(mb, vtype="B") # Add mb binary variables

# Add objective:  min a^T * y + b^T * z
M.addObjective("min", a@y + b@z)

# Construct matrix-expression:  sum H_j * y_j + sum G_j + z_j + D
mat_expr = sum(H[j]*y[j] for j in range(mc)) + \
           sum(G[j]*z[j] for j in range(mb)) + D

# Add PSD-constraint:  mat_expr > 0 (dual PSD-constraint)
M.addPSDConstraint(mat_expr)


M.writeCBF(file_name) # Write to file
```

Isometry

We solve randomly generated instances of

$$
\begin{aligned}
\min \quad & \langle A^\top A, X \rangle \\
\text{s.t.} \quad & \langle X \rangle = 1 \\
& \sum_{j=1}^{n} z_j \leq \kappa \\
& -\frac{1}{2} z_j \leq X_{ij} \leq \frac{1}{2} z_j, \qquad\qquad j = 1, ..., n, \ \forall i \\
& X \succeq 0 \\
& X \in \mathbb{R}^{n \times n},\ z \in \{0,1\}^n,
\end{aligned}
$$

where D is chosen such that the problem has at least one optimal solution.

```
def make_C(n):
    # return C: n x n numpy array C = A^T * A

def make_E(i, j, n):
    # return E_ij:  n x n numpy array
              E_ij = 0.5 * (e_i * e_j^T + e_j * e_i^T)

file_name = "temp_cbf.cbf"
n, kappa = 10, 3

M = Model()
C = make_C(n)
X = M.addPSDVar(n) # Add n x n PSD-variable (and constraint):  X > 0
z = M.addVars(n, vtype="B") # Add n variables:  z

obj_expr = C*X # construct linear expression:  <C,X>
M.addObjective("min", obj_expr) # Add objective:  min <C,X>


I = np.eye(n)
lin_expr = I*X # construct linear expression:  <I,X> = <X>
M.addConstraint(lin_expr == 1) # Add constraint:  <X> == 1


ones = np.ones(n)
lin_expr = ones@z # construct linear expression 1^T * z
M.addConstraint(lin_expr <= kappa) # Add constraint sum(z_j) <= kappa


# Add constraint -0.5 * z_j <= X_ij <= 0.5 * z_j, i = 1,..,n, j = 1,..,n
for j in range(n):
    z_j = z[j] # SingleVar z_j
    for i in range(n):
        E = make_E(i, j, n) # E_ij, s.t.  X_ij = <E_ij, X>
        M.addConstraint(E*X <= 0.5*z_j) # Add con:  X_ij <= 0.5 * z_j
        M.addConstraint(E*X >= -0.5*z_j) # Add con:  X_ij >= - 0.5 * z_j

M.writeCBF(file_name)
```

19 The CBF-format

Starting on the next page I have included my interpretation of how the CBF-format is written. First some comments:

Note on variable bounds:

- The CBF-format can handle variable bounds separately only if $x \in \mathcal{K}$ (see list below), but we still want to give the user the option to put in variable bounds:
- I have handled this by not utilizing the conic variable constraint at all – i.e., put all variables in F (i.e., $\mathbb{R}$), and add **all** variable bounds as affine constraints.
- For 'readability' the bound constraints are put separately in the CON-section. This means the last two lines are the lower- and upper bounds respectively
- I thought about fully grouping the constraints based on cone, but settled on writing the rest of them out in "input-order"

Note on indices:

- Previously in the PDF I have uses index-notation corresponding to ($j \in J$, $\ell \in L$):

  `'sum H[i,l]*x[l] + D[i]',  and  'sum <F[i,j],X[j]> + sum a[i,l]*x[l] + b[i]'`

  whereas here I will be using ($j \in J$, $j \in L$):

  `'sum H[i,j]*x[j] + D[i]',  and  'sum <F[i,j],X[j]> + sum a[i,j]*x[j] + b[i]'`

- i.e., index j corresponding to a Variable or SingleVar instance is not the same as index j corresponding to a PSDVar instance.

Relevant cones ($\mathcal{K}$):

- F : $x \in \mathbb{R}$
- L_+: $x \geq 0$
- L_-: $x \leq 0$
- $L_=$: $x = 0$

All values not specified in the CBF-file are assumed to be zero, so no zero-values will be specified.

```
# Commented out sections are not in the final file, they are comments
# Often '#' is used to clarify *type* (such as int/float) of below input-column
# All indents/added spaces are for readability, and are not in the final file
# Everything after a colon (:) is a comment, and is not in the final file

# File format
VER
2

   - - -

# Problem structure
OBJSENSE
# str
  MIN    : or MAX

   - - -

# PSD-variables
PSDVAR
# int
   J   : number of PSD-variables
# int
  n_1  : size n of PSD-var 1
  .
  .
  n_j  : size n of PSD-var j
  .
  .
  n_J  : sixe n of PSD-var J

   - - -

# Non-matrix variables
VAR
# int   int
  N     K    : number of variables (N), and number of conic domains they are restricted to (K)
# str   int
  c_1   n_1  : name "c" of cone 1 and number of variables n constricted to this cone
  .
  .
  c_k   n_k  : name "c" of cone k and number of variables n constricted to this cone
  .
  .
  c_K   n_K  : name "c" of cone K and number of variables n constricted to this cone
# Note: sum_k n_k = N

# Current implementation does not utilize this, instead adds all as linear constraints,
# i.e., only uses cone F. So this will always look like:
# VAR
# N 1 : Only one cone
# F N : All variables constricted to F-cone (i.e., R)

   - - -
```

```
# Integer variables (only non-matrix variables)
INT
# int
  N    : number of integer variables
# int
  j_1  : index j of integer variable 1
  .
  .
  j_n  : index j of integer variable n
  .
  .
  j_N  : index j of integer variable N

   - - -

PSDCON, (PSD constraints)
# int
  J    : number of PSD constraints in problem
# int
  n_1  : size n of matrices involved in PSD constraint 1
  .
  .
  n_j  : size n of matrices involved in PSD constraint j
  .
  .
  n_J  : size n of matrices involved in PSD constraint J

   - - -

CON, (affine constraints)
# int  int
  N    K      : number of scalar constraints (N) and number of cones they constrict to (K)
# str  int
  c_1  n_1    : name "c" of cone 1 and number of constraints n constricted to this cone
  .
  .
  c_k  n_k    : name "c" of cone k and number of constraints n constricted to this cone
  .
  .
  c_K  n_K    : name "c" of cone K and number of constraints n constricted to this cone
# Note: sum_k n_k = N

# For readability all bound constraints are put at the end here, i.e.,
# c_(K-1) = L+, n_(K-1) = number of LB-constraints
# c_(K)   = L-, n_(K)   = number of UB-constraints

   - - -
```

```
# Below problem data is specified (only for non-zero values)

   - - -

# Objective function:
# sum <OF[j], X[j]> + sum oa[j]*x[j] + ob
#  j    = variable index (either matrix or single variable)
# (k,l) = matrix element index

OBJFCOORD
# int
  N    : number of coordinates to be specified
# int int int float
  j   k   l   OF[j][k,l]  # Note: k >= l
  .
  .
# Total of N lines

OBJACOORD
# int
  N   : number of coordinates to be specified
# int  float
  j    oa[j]
  .
  .
# Total of N lines

OBJBCOORD
# float
  ob

   - - -

# Affine constraints:
# sum <F[i,j],X[j]>  +  sum  a[i,j]*x[j] + b[i] /in cone C, for all i
# the cone C is derives from information in 'CON'
#   i   = constraint index  (number of constraints comes from 'CON')
#   j   = variable index (either matrix or single variable)
# (k,l) = matrix element index


FCOORD
# int
  N   : number of coordinates to be specified
# int int int int float
  i   j   k   l   F[i,j][k,l]  # Note: k >= l
  .
  .
# Total of N lines
```

```
ACOORD
# int
  N   : number of coordinates to be specified
# int int  float
  i   j    a[i,j]
  .
  .
# Total of N lines
# The last 'n_(K-1) + n_(K)' lines are the LB and UB variable-constraints


BCOORD
# int
  N  : number of coordinates to be specified
# int  float
  i    b[i]
  .
  .
# Total of N lines

   - - -

# PSD-constraint:
# sum H[i,j]*x[j] + D[i] /in PSD-cone, for all i
#   i   = constraint index  (number of constraints comes from 'PSDCON')
#   j   = variable index (either matrix or single variable)
# (k,l) = matrix element index


HCOORD
# int
  N   : number of coordinates to be specified
# int int int int  float
  i   j   k   l    H[i,j][k,l]  # Note: k >= l
  .
  .
# Total of N lines


DCOORD
# int
  N   : number of coordinates to be specified
# int int int   float
   i   k   l    D[i][k,l]  # Note: k >= l
   .
   .
# Total of N lines

   - - -

# No implementation of CHANGE

   - F I N -
```