

Ncpol2sdpa Manual

Peter Wittek

ICFO-The Institute of Photonic Sciences and University of Borås

1 Introduction

Ncpol2sdpa is a tool to convert a polynomial optimization problem of noncommuting variables to a sparse semidefinite programming (SDP) problem that can be processed by the SDPA¹ family of solvers Yamashita et al. (2003). The optimization problem can be unconstrained or constrained by equalities and inequalities.

The objective is to be able to solve very large scale optimization problems. For example, a convergent series of lower bounds can be obtained for ground state problems with arbitrary Hamiltonians.

The implementation has an intuitive syntax for entering Hamiltonians and it scales for a larger number of noncommuting variables using a sparse representation of the SDP problem. The code is available in the Python Package Index at <https://pypi.python.org/pypi/ncpol2sdpa/> and the development version is at <http://peterwittek.github.io/ncpol2sdpa/>.

For further information on the internal workings of the library, please refer to corresponding paper (Wittek, 2014).

2 Dependencies and compilation

The implementation requires SymPy² $\geq 0.7.2$ (Joyner et al., 2012) and SciPy³ in the Python search path. The code is compatible with both Python 2 and 3, but using version 3 incurs a major decrease in performance. Follow the standard procedure for installing Python modules:

```
$ sudo pip install ncpol2sdpa
```

If you use the development version, install it from the source code:

```
$ sudo python setup.py install
```

¹<http://sdpa.sourceforge.net/>

²<http://sympy.org/>

³<http://scipy.org/>

3 Usage

The implementation follows an object-oriented design. The core object is `SdpRelaxation`. There are three steps to generate the relaxation:

1. Instantiate the `SdpRelaxation` object.
2. Get the relaxation.
3. Write the relaxation to a file or solve the problem.

The second step is the most time consuming, often running for hours as the number of noncommuting variables increases.

To instantiate the `SdpRelaxation` object, you need to specify the noncommuting variables:

```
X = ... # Define noncommuting variables
sdpRelaxation = SdpRelaxation(X)
```

Getting the relaxation also follows an almost identical syntax. It requires all the information about the polynomial optimization problem itself: the objective function, an associative array of the inequalities, equalities, the monomial substitutions, and also the level of the relaxation:

```
sdpRelaxation.get_relaxation(obj, inequalities, equalities,
                             monomial_substitution, level)
```

The last step in is to write out the relaxation to a sparse SDPA file. The method (`write_to_sdpa`) takes one parameter, the file name. Alternatively, if SDPA is in the search path, then it can be solved by invoking a helper function (`solve_sdp`). Alternatively, MOSEK⁴ is also supported for writing a problem and solving it. Using a compatibility layer to PICOS⁵, it is also possible to solve the problem with a range of other solvers, including CVXOPT⁶.

4 Examples

4.1 Example 1: Toy example

Consider the following polynomial optimization problem (Pironio et al., 2010):

$$\min_{x \in \mathbb{R}^2} x_1 x_2 + x_2 x_1$$

such that

$$-x_2^2 + x_2 + 0.5 \geq 0$$

⁴<http://www.mosek.com/>

⁵<http://picos.zib.de/>

⁶<http://cvxopt.org/>

$$x_1^2 - x_1 = 0.$$

Entering the objective function and the inequality constraint is easy. The equality constraint is a simple projection. We either substitute two inequalities to replace the equality, or treat the equality as a monomial substitution. The second option leads to a sparser SDP relaxation. The code samples below take this approach. In this case, the monomial basis is $\{1, x_1, x_2, x_1x_2, x_2x_1, x_2^2\}$. The corresponding relaxation is written as

$$\min_y y_{12} + y_{21}$$

such that

$$\begin{bmatrix} 1 & y_1 & y_2 & y_{12} & y_{21} & y_{22} \\ y_1 & y_1 & y_{12} & y_{12} & y_{121} & y_{122} \\ y_2 & y_{21} & y_{22} & y_{212} & y_{221} & y_{222} \\ y_{21} & y_{21} & y_{212} & y_{212} & y_{2121} & y_{2122} \\ y_{12} & y_{121} & y_{122} & y_{1212} & y_{1221} & y_{1222} \\ y_{22} & y_{221} & y_{222} & y_{2212} & y_{2221} & y_{2222} \end{bmatrix} \succeq 0$$

$$\begin{bmatrix} -y_{22} + y_2 + 0.5 & -y_{221} + y_{21} + 0.5y_1 & -y_{222} + y_{22} + 0.5y_2 \\ -y_{221} + y_{21} + 0.5y_1 & -y_{1221} + y_{121} + 0.5y_1 & -y_{1222} + y_{122} + 0.5y_{12} \\ -y_{222} + y_{22} + 0.5y_2 & -y_{1222} + y_{122} + 0.5y_{12} & -y_{2222} + y_{222} + 0.5y_{22} \end{bmatrix} \succeq 0.$$

Apart from the matrices being symmetric, notice other regular patterns between the elements. These are taken care of as additional constraints in the implementation. The optimum for the objective function is $-3/4$. The implementation reads as follows:

```
from ncpol2sdpa import generate_variables, SdpRelaxation, write_to_sdpa

# Number of Hermitian variables
n_vars = 2
# Level of relaxation
level = 2

# Get Hermitian variables
X = generate_variables(n_vars, hermitian=True)

# Define the objective function
obj = X[0] * X[1] + X[1] * X[0]

# Inequality constraints
inequalities = [-X[1] ** 2 + X[1] + 0.5]

# Equality constraints
```

```

equalities = []

# Simple monomial substitutions
monomial_substitution = {}
monomial_substitution[X[0] ** 2] = X[0]

# Obtain SDP relaxation
sdpRelaxation = SdpRelaxation(X)
sdpRelaxation.get_relaxation(obj, inequalities, equalities,
                             monomial_substitution, level)
write_to_sdpa(sdpRelaxation, 'example_noncommutative.dat-s')

```

Any flavour of the SDPA family of solvers will solve the exported problem:

```
$ sdpa examplenc.dat-s examplenc.out
```

If the SDPA solver is in the search path, we can invoke the solver from Python:

```

from ncpol2sdpa import solve_sdp
primal, dual = solve_sdp(sdpRelaxation)

```

The relevant part of the output shows the optimum for the objective function:

```

objValPrimal = -7.5000001721851994e-01
objValDual   = -7.5000007373829902e-01

```

This is close to the analytical optimum of $-3/4$.

4.2 Example 2: Using MOSEK or PICOS

Apart from SDPA, MOSEK also enjoys full support. Using the preliminaries of the problem outlined in Section 4.1, once we have the relaxation, we can convert it to a MOSEK task and solve it:

```

task = convert_to_mosek(sdpRelaxation)
task.optimize()
task.solutionsummary(mosek.streamtype.msg)

```

Please ensure that the MOSEK is operational.

A compatibility layer with PICOS allows calling a wider range of solvers. Assuming that the PICOS dependencies are in PYTHONPATH, we can pass an argument to the function `get_relaxation` to generate a PICOS optimization problem. Using the same example as before, we change the relevant function call to:

```

P = sdpRelaxation.get_relaxation(obj, inequalities, equalities,
                                 monomial_substitution, level, target='picos')

```

This returns a PICOS problem, and with that, we can solve it with any solver that PICOS supports:

```
P.solve()
```

4.3 Example 3: Bosonic system

A more sophisticated application is also supplied with the code (`test-harmonic_oscillator.py`), which implements the Hamiltonian of a bosonic system on a 1D line. Since it uses non-Hermitian variables, a C++ implementation is currently not feasible.

The system Hamiltonian describes N harmonic oscillators with a parameter ω . It is the result of second quantization and it is subject to bosonic constraints on the ladder operators a_k and a_k^\dagger (see, for instance, Section 22.2 in Fayngold and Fayngold (2013)). The Hamiltonian is written as

$$H = \hbar\omega \sum_i \left(a_i^\dagger a_i + \frac{1}{2} \right). \quad (1)$$

Here \dagger stands for the adjoint operation. The constraints on the ladder operators are given as

$$\begin{aligned} [a_i, a_j^\dagger] &= \delta_{ij} \\ [a_i, a_j] &= 0 \\ [a_j^\dagger, a_j^\dagger] &= 0, \end{aligned} \quad (2)$$

where $[\cdot, \cdot]$ stands for the commutation operator $[a, b] = ab - ba$.

Clearly, most of the constraints are monomial substitutions, except $[a_i, a_i^\dagger] = 1$, which needs to be defined as an equality. The Python code for generating the SDP relaxation is provided below. We set $\omega = 1$, and we also set Planck's constant \hbar to one, to obtain numerical results that are easier to interpret.

```
from sympy.physics.quantum.dagger import Dagger
from ncpol2sdpa import generate_variables, \
    bosonic_constraints, \
    SdpRelaxation, write_to_sdp

# level of relaxation
level = 1

# Number of variables
N = 4

# Parameters for the Hamiltonian
hbar, omega = 1, 1

# Define ladder operators
```

```

a = generate_variables(N, name='a')

hamiltonian = 0
for i in range(N):
    hamiltonian += hbar*omega*(Dagger(a[i])*a[i]+0.5)

monomial_substitutions, equalities = bosonic_constraints(a)
inequalities = []

time0 = time.time()
#Obtain SDP relaxation
print("Obtaining SDP relaxation...")
verbose = 1
sdpRelaxation = SdpRelaxation(a)
sdpRelaxation.get_relaxation(hamiltonian, inequalities, equalities,
                             monomial_substitutions, level, verbose)
#Export relaxation to SDPA format
print("Writing to disk...")
write_to_sdpa(sdpRelaxation, 'harmonic_oscillator.dat-s')

```

Solving the SDP for $N = 4$, for instance, gives the following result:

```

objValPrimal = +1.9999998358414430e+00
objValDual   = +1.9999993671869802e+00

```

This is very close to the analytic result of 2. The result is similarly precise for arbitrary numbers of oscillators.

It is remarkable that we get the correct value at the first level of relaxation, but this property is typical for bosonic systems (Navascués et al., 2013).

References

- Fayngold, M. and Fayngold, V. (2013). *Quantum Mechanics and Quantum Information*. Wiley-VCH.
- Joyner, D., Čertík, O., Meurer, A., and Granger, B. E. (2012). Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra*, 45(3/4):225–234.
- Navascués, M., García-Sáez, A., Acín, A., Pironio, S., and Plenio, M. B. (2013). A paradox in bosonic energy computations via semidefinite programming relaxations. *New Journal of Physics*, 15(2): 023026.
- Pironio, S., Navascues, M., and Acín, A. (2010). Convergent relaxations of polynomial optimization problems with noncommuting variables. *SIAM Journal on Optimization*, 20(5):2157–2180.

Wittek, P. (2014). Ncpol2sdpa – Sparse Semidefinite Programming Relaxations for Polynomial Optimization Problems of Noncommuting Variables. *To appear in the ACM Transactions on Mathematical Software*.

Yamashita, M., Fujisawa, K., and Kojima, M. (2003). SDPARA: Semidefinite programming algorithm parallel version. *Parallel Computing*, 29(8):1053–1067.