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 SDPA[1] 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 SymPy[2]$\geq$ 0.7.2 (Joyner et al., 2012) and SciPy[3] 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, MOSEK[4] is also supported for writing a problem and solving it. Using a compatibility layer to PICOS[5], it is also possible to solve the problem with a range of other solvers, including CVXOPT[6].

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

$$\left[\begin{array}{c|c|cc|cc} 1 & y_1 & y_2 & y_{12} & y_{21} & y_{22} \\ \hline y_1 & y_1 & y_{12} & y_{12} & y_{121} & y_{122} \\ y_2 & y_{21} & y_{22} & y_{212} & y_{221} & y_{222} \\ \hline y_{21} & y_{21} & y_{212} & y_{212} & y_{2121} & y_{2122} \\ y_{12} & y_{121} & y_{122} & y_{1212} & y_{1221} & y_{1222} \\ y_{22} & y_{221} & y_{222} & y_{2212} & y_{2221} & y_{2222} \end{array}\right] \succeq 0$$

$$\left[\begin{array}{c|cc} -y_{22} + y_2 + 0.5 & -y_{221} + y_{21} + 0.5y_1 & -y_{222} + y_{22} + 0.5y_2 \\ \hline -y_{221} + y_{21} + 0.5y_1 & -y_{1221} + y_{121} + 0.5y_1 & -y_{1222} + y_{122} + 0.5y_{12} \\ -y_{222} + y_{22} + 0.5y_2 & -y_{1222} + y_{122} + 0.5y_{12} & -y_{2222} + y_{222} + 0.5y_{22} \end{array}\right] \succeq 0.$$

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 ranger 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

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

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

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

where $[.,.]$ 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()

print("Obtaining SDP relaxation...")
verbose = 1
sdpRelaxation = SdpRelaxation(a)
sdpRelaxation.get_relaxation(hamiltonian, inequalities, equalities,
                             monomial_substitutions, level,
                             removeequalities=True)
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).

4.4 Example 4: Using the Nieto-Silleras hierarchy

One of the newer approaches to the SDP relaxations takes all joint probabilities into consideration when looking for a maximum guessing probability, and not just the ones included in a particular Bell inequality (Nieto-Silleras et al., 2014; Bancal et al., 2014). Ncpol2sdpa can generate the respective hierarchy.

To deal with the joint probabilities necessary for setting constraints, we also rely on QuTiP (Johansson et al., 2013):

```
from math import sqrt
from qutip import tensor, basis, sigmax, sigmay, expect, qeye
from ncpol2sdpa import SdpRelaxation, flatten, solve_sdp, \
                       generate_measurements, \
                       projective_measurement_constraints
```

We will work in a CHSH scenario where we are trying to find the maximum guessing probability of the first projector of Alice's first measurement. We generate the joint probability distribution on the maximally entangled state with the measurements that give the maximum quantum violation of the CHSH inequality:

```
def joint_probabilities():
    psi = (tensor(basis(2,0),basis(2,0)) +
           tensor(basis(2,1),basis(2,1))).unit()
    A_0 = sigmax()
    A_1 = sigmay()
    B_0 = (-sigmay()+sigmax())/sqrt(2)
    B_1 = (sigmay()+sigmax())/sqrt(2)

    A_00 = (qeye(2) + A_0)/2
    A_10 = (qeye(2) + A_1)/2
    B_00 = (qeye(2) + B_0)/2
    B_10 = (qeye(2) + B_1)/2

    p=[]
    p.append(expect(tensor(A_00, qeye(2)), psi))
    p.append(expect(tensor(A_10, qeye(2)), psi))
    p.append(expect(tensor(qeye(2), B_00), psi))
    p.append(expect(tensor(qeye(2), B_10), psi))

    p.append(expect(tensor(A_00, B_00), psi))
    p.append(expect(tensor(A_00, B_10), psi))
    p.append(expect(tensor(A_10, B_00), psi))
    p.append(expect(tensor(A_10, B_10), psi))
    return p
```

Next we need the basic configuration of the projectors. We also set the level of the SDP relaxation and the objective.

```
level = 1
A_configuration = [2, 2]
B_configuration = [2, 2]
P_A = generate_measurements(A_configuration, 'P_A')
P_B = generate_measurements(B_configuration, 'P_B')
monomial_substitutions = projective_measurement_constraints(
    P_A, P_B)
objective = -P_A[0][0]
```

We must define further constraints, namely that the joint probabilities must match:

```
probabilities = joint_probabilities()
equalities = []
k=0
for i in range(len(A_configuration)):
    equalities.append(P_A[i][0] - probabilities[k])
    k += 1
for i in range(len(B_configuration)):
```

```
    equalities.append(P_B[i][0] - probabilities[k])
    k += 1
for i in range(len(A_configuration)):
    for j in range(len(B_configuration)):
        equalities.append(P_A[i][0]*P_B[j][0] - probabilities[k])
        k += 1
```

From here, the solution follows the usual pathway, indicating that we are requesting the Nieto-Silleras hierarchy:

```
sdpRelaxation = SdpRelaxation([flatten([P_A, P_B])], verbose=2,
                              hierarchy="nieto-silleras")
sdpRelaxation.get_relaxation(objective, [], equalities,
                             monomial_substitutions, level)

print(solve_sdp(sdpRelaxation))
```

4.5 Example 5: Using the Moroder hierarchy

This type of hierarchy allows for a wider range of constraints of the optimization problems, including ones that are not of polynomial form (Moroder et al., 2013). These constraints are hard to impose using SymPy and the sparse structures in Ncpol2Sdpa. For this reason, we separate two steps: generating the SDP and post-processing the SDP to impose extra constraints. This second step can be done in MATLAB, for instance. We need to import a slightly different set of functions:

```
from ncpol2sdpa import SdpRelaxation, flatten, write_to_sdpa, \
                       generate_measurements, \
                       projective_measurement_constraints, \
                       define_objective_with_I
```

Then we set up the problem with specifically with the CHSH inequality in the probability picture as the objective function:

```
level = 1
A_configuration = [2, 2]
B_configuration = [2, 2]
I = [[ 0,   -1,    0 ],
     [-1,    1,    1 ],
     [ 0,    1,   -1 ]]
A = generate_measurements(A_configuration, 'A')
B = generate_measurements(B_configuration, 'B')
monomial_substitutions = projective_measurement_constraints(A, B)
objective = define_objective_with_I(I, A, B)
```

When obtaining the relaxation for this kind of problem, it can prove useful to disable the normalization of the top-left element of the moment matrix. Naturally, before solving the problem this should be set to zero, but further processing

of the SDP matrix can be easier without this constraint set a priori. Hence we write:

```
sdpRelaxation = SdpRelaxation([flatten(A), flatten(B)], verbose=2,
                              hierarchy="moroder", normalized=False)
sdpRelaxation.get_relaxation(objective, [], [],
                             monomial_substitutions, level)
write_to_sdpa(sdpRelaxation, "chsh-moroder.dat-s")
```

For instance, reading this file with SeDuMi's `fromsdpa` function (Sturm, 1999), we can impose the positivity of the partial trace of the moment matrix, or decompose the moment matrix in various forms.

References

Bancal, J.-D., Sheridan, L. and Scarani, V. (2014). More randomness from the same data. *New Journal of Physics*, 16(3):033011.

Fayngold, M. and Fayngold, V. (2013). *Quantum Mechanics and Quantum Information*. Wiley-VCH.

Johansson, J., Nation, P. and Nori, F. (2013). A Python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184(4):1234–1240

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.

Moroder, T., Bancal, J.-D., Liang, Y.-C., Hofmann, M. and G̈hne, O. (2013). Device-independent entanglement quantification and related applications *Physics Review Letters*, 111(3):030501.

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.

Nieto-Silleras, O., Pironio, S. and Silman, J. (2014) Using complete measurement statistics for optimal device-independent randomness evaluation. *New Journal of Physics*, 16(1):013035.

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.

Sturm, J. (2010). Using SeDuMi 1.02, a MATLAB Toolbox for Optimization Over Symmetric Cones. *Optimization Methods and Software*, 11(1-4):625–653.

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.