Ncpol2sdpa Documentation

Release 1.12.1

Peter Wittek

Mar 16, 2017

Contents

1 Introduction **1**
- 1.1 Copyright and License . . . 2
- 1.2 Acknowledgment . . . 2

2 Download and Installation **3**
- 2.1 Dependencies . . . 3
- 2.2 Installation . . . 3

3 Tutorial **5**
- 3.1 Defining a Polynomial Optimization Problem of Commuting Variables . . . 5
- 3.2 Generating and Solving the Relaxation . . . 6
- 3.3 Analyzing the Solution . . . 7
- 3.4 Debugging the SDP Relaxation . . . 7
- 3.5 Defining and Solving an Optimization Problem of Noncommuting Variables . . . 8

4 Examples **9**
- 4.1 Example 1: Max-cut . . . 9
- 4.2 Example 2: Parametric Polynomial Optimization Problems . . . 9
- 4.3 Example 3: Sparse Relaxation with Chordal Extension . . . 11
- 4.4 Example 4: Mixed-Level Relaxation of a Bell Inequality . . . 11
- 4.5 Example 5: Additional manipulation of the generated SDPs with PICOS . . . 12
- 4.6 Example 6: Bosonic System . . . 13
- 4.7 Example 7: Using the Nieto-Silleras Hierarchy . . . 13
- 4.8 Example 8: Using the Moroder Hierarchy . . . 14

5 Revision History **17**

6 Function Reference **23**
- 6.1 SdpRelaxation Class . . . 23
- 6.2 MoroderHierarchy Class . . . 27
- 6.3 SteeringHierarchy Class . . . 30
- 6.4 FaacetsRelaxation Class . . . 34
- 6.5 Functions to Help Define Polynomial Optimization Problems . . . 34
- 6.6 Functions to Study Output of Solver . . . 35
- 6.7 Functions and Classes to Define Physics Problems . . . 36

7 References **39**

CHAPTER 1

Introduction

Ncpol2sdpa solves global polynomial optimization problems of either commutative variables or noncommutative operators through a semidefinite programming (SDP) relaxation. The optimization problem can be unconstrained or constrained by equalities and inequalities, and also by constraints on the moments. The objective is to be able to solve large scale optimization problems. Example applications include:

- When the polynomial optimization problem is defined over commutative variables, the generated SDP hierarchy is identical to Lasserre's. In this case, the functionality resembles the MATLAB toolboxes Gloptipoly, and, with the chordal extension, SparsePOP.
- Relaxations of parametric and bilevel polynomial optimization problems.
- When the polynomials are over noncommutative operators, the generated SDP is a step in the Navascués-Pironio-Acín (NPA) hierarchy. The most notable example is calculating the maximum quantum violation of Bell inequalities, also in multipartite scenarios.
- Nieto-Silleras hierarchy for quantifying randomness and for calculating maximum guessing probability.
- Moroder hierarchy to enable PPT-style and other additional constraints.
- Sums-of-square (SOS) decomposition based on the dual solution.
- Ground-state energy problems: bosonic and fermionic systems, Pauli spin operators. This methodology closely resembles the reduced density matrix (RDM) method.
- Hierarchy for quantum steering.

The implementation has an intuitive syntax for entering problems and it scales for a larger number of noncommutative variables using a sparse representation of the SDP problem. Further details are found in the following paper:

- Peter Wittek. Algorithm 950: Ncpol2sdpa—Sparse Semidefinite Programming Relaxations for Polynomial Optimization Problems of Noncommuting Variables. *ACM Transactions on Mathematical Software*, 41(3), 21, 2015. DOI: 10.1145/2699464. arXiv:1308.6029.

Copyright and License

Acknowledgment

This work is supported by the European Commission Seventh Framework Programme under Grant Agreement Number FP7-601138 PERICLES, by the Red Espanola de Supercomputacion grants number FI-2013-1-0008 and FI-2013-3-0004, and by the Swedish National Infrastructure for Computing projects SNIC 2014/2-7 and SNIC 2015/1-162.

CHAPTER 2

Download and Installation

The package is available in the Python Package Index. The latest development version is available on GitHub.

Dependencies

The implementation requires SymPy and Numpy. The code is compatible with both Python 2 and 3, but using version 3 incurs a major decrease in performance.

While the default CPython interpreter is sufficient for small to medium-scale problems, execution time becomes excessive for larger problems. The code is compatible with Pypy. Using it yields a 10-20x speedup. If you use Pypy, you will need the Pypy fork of Numpy.

By default, Ncpol2sdpa does not require a solver, but then it will not be able to solve a generated relaxation either. Install any supported solver and it will be detected automatically.

Optional dependencies include:

- SDPA is a possible target solver.
- SciPy yields faster execution with the default CPython interpreter.
- PICOS is necessary for using the Cvxopt solver and for converting the problem to a PICOS instance.
- MOSEK Python module is necessary to work with the MOSEK solver.
- CVXPY is required for converting the problem to or by solving it by CVXPY.
- Cvxopt is required by both Chompack and PICOS.
- Chompack improves the sparsity of the chordal graph extension.

Installation

Follow the standard procedure for installing Python modules:

```
$ pip install ncpol2sdpa
```

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

```
$ git clone https://github.com/peterwittek/ncpol2sdpa.git
$ cd ncpol2sdpa
$ python setup.py install
```

CHAPTER 3

Tutorial

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

- Instantiate the SdpRelaxation object.
- Get the relaxation.
- 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 variables increases. Once the solution is obtained, it can be studied further with some helper functions.

To instantiate the SdpRelaxation object, you need to specify the variables. You can use any SymPy symbolic variable, as long as the adjoint operator is well-defined. The library also has helper functions to generate commutative or noncommutative variables or operators.

Getting the relaxation requires at least the level of relaxation, and the matching method, *SdpRelaxation.get_relaxation*, will generate the moment matrix. Additional elements of the problem, such as the objective function, inequalities, equalities, and constraints on the moments.

The last step in is to either solve or export the relaxation. The function *solve_sdp* or the class method *SdpRelaxation.solve* autodetects the possible solvers: SDPA, MOSEK, and CVXOPT. Alternatively, the method `write_to_file` exports the file to sparse SDPA format, which can be solved externally on a supercomputer, in MATLAB, or by any other means that accepts this input format.

Defining a Polynomial Optimization Problem of Commuting Variables

Consider the following polynomial optimization problem:

$$\min_{x \in \mathbb{R}^2} 2x_1x_2$$

such that

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

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

The equality constraint is a simple projection. We either substitute it with two inequalities 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 $\{1, x_1, x_2, x_1x_2, x_2^2\}$. The corresponding level-2 relaxation is written as

$$\min_y 2y_{12}$$

such that

$$\left[\begin{array}{c|cc|cc} 1 & y_1 & y_2 & y_{12} & y_{22} \\ \hline y_1 & y_1 & y_{12} & y_{12} & y_{122} \\ y_2 & y_{12} & y_{22} & y_{122} & y_{222} \\ \hline y_{12} & y_{12} & y_{122} & y_{122} & y_{1222} \\ y_{22} & y_{122} & y_{222} & y_{1222} & y_{2222} \end{array}\right] \succeq 0$$

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

Apart from the matrices being symmetric, notice other regular patterns between the elements – these are recognized in the relaxation and the same SDP variables are used for matching moments. To generate the relaxation, first we set up a few helper variables, including the symbolic variables used to define the polynomial objective function and constraint. The symbolic manipulations are based on SymPy.

```
from ncpol2sdpa import *

n_vars = 2 # Number of variables
level = 2  # Requested level of relaxation
x = generate_variables('x', n_vars)
```

By default, the generated variables are commutative. Alternatively, you can use standard SymPy symbols, but it is worth declaring them as real. With these variables, we can define the objective and the inequality constraint.

```
obj = x[0]*x[1] + x[1]*x[0]
inequalities = [-x[1]**2 + x[1] + 0.5>=0]
```

We can also write all inequality-type constraints assuming to be in the form ≥ 0 as

```
inequalities = [-x[1]**2 + x[1] + 0.5]
```

This is more convenient when we have a large number of constraints.

The equality, as discussed, is entered as a substitution rule:

```
substitutions = {x[0]**2 : x[0]}
```

Generating and Solving the Relaxation

After we defined the problem, we need to initialize the SDP relaxation object with the variables, and request generating the relaxation given the constraints:

```
sdp = SdpRelaxation(x)
sdp.get_relaxation(level, objective=obj, inequalities=inequalities,
                   substitutions=substitutions)
```

For large problems, getting the relaxation can take a long time. Once we have the relaxation, we can try to solve it solve it. Currently three solvers are supported fully: SDPA, MOSEK, and CVXOPT. If any of them are available, we obtain the solution by calling the `solve` method:

```
sdp.solve()
print(sdp.primal, sdp.dual, sdp.status)
```

This gives a solution close to the optimum around -0.7321. The solution and some status information and the time it takes to solve it become part of the relaxation object.

If no solver is detected, or you want more control over the parameters of the solver, or you want to solve the problem in MATLAB, you export the relaxation to SDPA format:

```
sdp.write_to_file('example.dat-s')
```

You can also specify a solver if you wish. For instance, if you want to use the arbitrary-precision solver that you have available in the path, along with a matching parameter file, you can call

```
sdp.solve(solver='sdpa', solverparameters={"executable":"sdpa_gmp",
                                           "paramsfile"="params.gmp.sdpa"})
```

If you have multiple solvers available, you might want to specify which exactly you want to use. For CVXOPT, call

```
sdp.solve(solver='cvxopt')
print(sdp.primal, sdp.dual)
```

This solution also requires PICOS on top of CXOPT. Alternatively, if you have MOSEK installed and it is callable from your Python distribution, you can request to use it:

sdp.solve(solver='mosek') print(sdp.primal, sdp.dual)

Analyzing the Solution

We can study individual entries of the solution matrix by providing the monomial we are interested in. For example:

```
sdp[X[0]*X[1]]
```

The sums-of-square (SOS) decomposition is extracted from the dual solution:

```
sigma = sdp.get_sos_decomposition()
```

If we solve the SDP with the arbitrary-precision solver `sdpa_gmp`, we can find a rank loop at level two, indicating that convergence has been achieved.

```
sdp.solve(solver='sdpa', solverparameters={"executable":"sdpa_gmp",
                                           "paramsfile"="params.gmp.sdpa"})
sdp.find_solution_ranks()
```

The output for this problem is `[2, 3]`, not showing a rank loop at this level of relaxation.

Debugging the SDP Relaxation

It often happens that solving a relaxation does not yield the expected results. To help understand what goes wrong, Ncpol2sdpa provides a function to write the relaxation in a comma separated file, in which the individual cells contain the respective monomials. The first line of the file is the objective function.

```
sdp.write_to_file("examples.csv")
```

Furthermore, the library can write out which SDP variable corresponds to which monomial by calling

```
sdp.save_monomial_index("monomials.txt")
```

Defining and Solving an Optimization Problem of Noncommuting Variables

Consider a slight variation of the problem discussed in the previous sections: change the algebra of the variables from commutative to Hermitian noncommutative, and use the following objective function:

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

The constraints remain identical:

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

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

Defining the problem, generating the relaxation, and solving it follow a similar pattern, but we request operators instead of variables.

```
X = generate_operators('X', n_vars, hermitian=True)
obj_nc = X[0] * X[1] + X[1] * X[0]
inequalities_nc = [-X[1] ** 2 + X[1] + 0.5]
substitutions_nc = {X[0]**2 : X[0]}
sdp_nc = SdpRelaxation(X)
sdp_nc.get_relaxation(level, objective=obj_nc, inequalities=inequalities_nc,
                      substitutions=substitutions_nc)
sdp_nc.solve()
```

This gives a solution very close to the analytical -3/4. Let us export the problem again:

```
sdp.write_to_file("examplenc.dat-s")
```

Solving this with the arbitrary-precision solver, we discover a rank loop:

```
sdp.solve(solver='sdpa', solverparameters={"executable":"sdpa_gmp",
                                           "paramsfile"="params.gmp.sdpa"})
sdp.find_solution_ranks()
```

The output is `[2, 2]`, indicating a rank loop and showing that the noncommutative case of the relaxation converges faster.

CHAPTER 4

Examples

Example 1: Max-cut

This is a polynomial optimization problem of commutative variables mentioned in Section 5.12 of Henrion et. al (2009). We rely on NumPy to remove the equality constraints from the problem

```
import numpy as np
W = np.diag(np.ones(8), 1) + np.diag(np.ones(7), 2) + np.diag([1, 1], 7) + \
    np.diag([1], 8)
W = W + W.T
n = len(W)
e = np.ones(n)
Q = (np.diag(np.dot(e.T, W)) - W) / 4

x = generate_variables('x', n)
equalities = [xi ** 2 - 1 for xi in x]

objective = -np.dot(x, np.dot(Q, np.transpose(x)))

sdp = SdpRelaxation(x)
sdp.get_relaxation(1, objective=objective, equalities=equalities,
                   removeequalities=True)
sdp.solve()
```

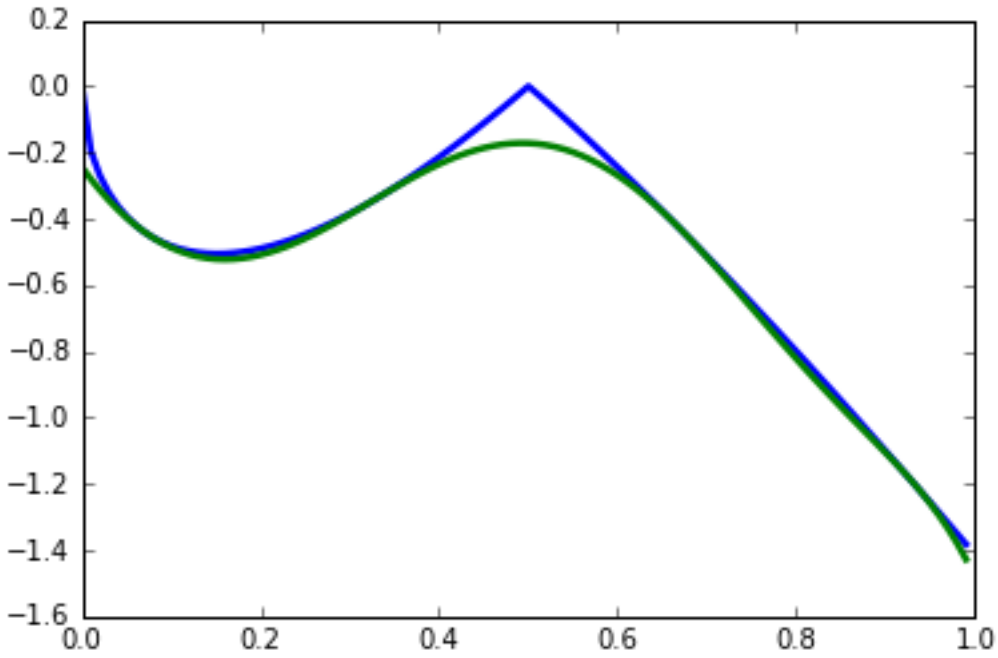
Example 2: Parametric Polynomial Optimization Problems

In a parametric polynomial optimization problem, we can separate two sets of variables, and one set acts as a parameter to the problem. More formally, we would like to find the following function:

$$J(x) = \inf_{y \in \mathbb{R}^m} \{f(x,y) : h_j(y) \geq 0, j = 1, \ldots, r\}$$

where $x \in \mathbf{X} = \{x \in \mathbb{R}^n : h_k(x) \geq 0, k = r+1, \ldots, t\}$. We can approximate $J(x)$ using the dual an SDP relaxation. The following implements Example 4 from Lasserre (2010).

```
from math import sqrt
from sympy import integrate, N
import matplotlib.pyplot as plt

def J(x):
    return -2*abs(1-2*x)*sqrt(x/(1+x))

def Jk(x, coeffs):
    return sum(ci*x**i for i, ci in enumerate(coeffs))

level = 4
x = generate_variables('x')[0]
y = generate_variables('y', 2)
f = (1-2*x)*(y[0] + y[1])

gamma = [integrate(x**i, (x, 0, 1)) for i in range(1, 2*level+1)]
marginals = flatten([[x**i-N(gamma[i-1]), N(gamma[i-1])-x**i]
                     for i in range(1, 2*level+1)])

inequalities = [x*y[0]**2 + y[1]**2 - x,  - x*y[0]**2 - y[1]**2 + x,
                y[0]**2 + x*y[1]**2 - x,  - y[0]**2 - x*y[1]**2 + x,
                1-x, x]
sdp = SdpRelaxation(flatten([x, y]))
sdp.get_relaxation(level, objective=f, momentinequalities=marginals,
                   inequalities=inequalities)
sdp.solve()
coeffs = [sdp.extract_dual_value(0, range(len(inequalities)+1))]
coeffs += [sdp.y_mat[len(inequalities)+1+2*i][0][0] - sdp.y_
↪mat[len(inequalities)+1+2*i+1][0][0]
           for i in range(len(marginals)//2)]

x_domain = [i/100. for i in range(100)]
plt.plot(x_domain, [J(xi) for xi in x_domain], linewidth=2.5)
plt.plot(x_domain, [Jk(xi, coeffs) for xi in x_domain], linewidth=2.5)
plt.show()
```



Example 3: Sparse Relaxation with Chordal Extension

This method replicates the behaviour of SparsePOP (Waki et. al, 2008). The following is a simple example:

```
level = 2
X = generate_variables('x', 3)

obj = X[1] - 2*X[0]*X[1] + X[1]*X[2]
inequalities = [1-X[0]**2-X[1]**2, 1-X[1]**2-X[2]**2]

sdp = SdpRelaxation(X)
sdp.get_relaxation(level, objective=obj, inequalities=inequalities,
                   chordal_extension=True)
sdp.solve()
print(sdp.primal, sdp.dual)
```

Example 4: Mixed-Level Relaxation of a Bell Inequality

It is often the case that moving to a higher-order relaxation is computationally prohibitive. For these cases, it is possible to inject extra monomials to a lower level relaxation. We refer to this case as a mixed-level relaxation.

As an example, we consider the CHSH inequality in the probability picture at level 1+AB relaxation. The lazy way of doing this is as follows:

```
level = 1
I = [[ 0,   -1,    0 ],
     [-1,    1,    1 ],
     [ 0,    1,   -1 ]]
print(maximum_violation(A_configuration, B_configuration, I, level,
      extra='AB')
```

This will immediately give you the negative of the maximum violation. The function *maximum_violation* only works for two-party configuration, so for educational purposes, we spell out what goes on in the background. With *level* and *I* defined as above, we create the measurements that will make up the probabilities, and define the objective function with the *I* matrix.

```
P = Probability([2, 2], [2, 2])
objective = define_objective_with_I(I, P)
```

Unfortunately, the function *define_objective_with_I* only works for two parties again, which is not surprising, as it would be hard to define an *I* matrix for more than two parties. So if you have a multipartite scenario, you can use the probabilities to define your Bell inequality. For the CHSH, it is

```
CHSH = -P([0],[0],'A') + P([0,0],[0,0])  + P([0,0],[0,1]) + \
        P([0,0],[1,0]) - P([0,0],[1,1]) - P([0],[0],'B')
```

Note that we can only minimize a function, so we have to flip the sign to get the same objective function as above:

```
objective = -CHSH
```

We need to generate the monomials we would like to add to the relaxation. This is aided by a helper function in the class *Probability*. We only need to provide the strings we would like to see – this time it is AB:

```
sdp = SdpRelaxation(P.get_all_operators())
sdp.get_relaxation(level, objective=objective,
                   substitutions=P.substitutions,
                   extramonomials=P.get_extra_monomials('AB'))
sdp.solve()
print(sdp.primal)
```

Example 5: Additional manipulation of the generated SDPs with PICOS

A compatibility layer with PICOS allows additional manipulations of the optimization problem and also 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 = sdp.convert_to_picos()
```

This returns a PICOS problem. For instance, we can manually define the value of certain elements of the moment matrix before solving the SDP:

```
X = P.get_variable('X')
P.add_constraint(X[0, 1] == 0.5)
```

Finally we can solve the SDP with any of solvers that PICOS supports:

```
P.solve()
```

Example 6: Bosonic System

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 M. Fayngold and Fayngold (2013)). The Hamiltonian is written as

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

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

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

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

level = 1          # Level of relaxation
N = 4              # Number of variables
hbar, omega = 1, 1 # Parameters for the Hamiltonian

# Define ladder operators
a = generate_operators('a', N)

hamiltonian = sum(hbar*omega*(Dagger(ai)*ai+0.5) for ai in a)
substitutions = bosonic_constraints(a)

sdp = SdpRelaxation(a)
sdp.get_relaxation(level, objective=hamiltonian,
                   substitutions=substitutions)
sdp.solve()
```

The result is very close to two. 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).

Example 7: 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, Pironio, and Silman 2014; Bancal, Sheridan, and Scarani 2014). Ncpol2sdpa can generate the respective hierarchy.

To deal with the joint probabilities necessary for setting constraints, we also rely on QuTiP (Johansson, Nation, and Nori 2013):

```
from math import sqrt
from qutip import tensor, basis, sigmax, sigmay, expect, qeye
```

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:

```
psi = (tensor(basis(2,0),basis(2,0)) + tensor(basis(2,1),basis(2,1))).unit()
A = [(qeye(2) + sigmax())/2, (qeye(2) + sigmay())/2]
B = [(qeye(2) + (-sigmay()+sigmax())/sqrt(2))/2,
     (qeye(2) + (sigmay()+sigmax())/sqrt(2))/2]
```

Next we need the basic configuration of the probabilities and we must make them match the observed distribution.

```
P = Probability([2, 2], [2, 2])
behaviour_constraints = [
  P([0],[0],'A')-expect(tensor(A[0], qeye(2)), psi),
  P([0],[1],'A')-expect(tensor(A[1], qeye(2)), psi),
  P([0],[0],'B')-expect(tensor(qeye(2), B[0]), psi),
  P([0],[1],'B')-expect(tensor(qeye(2), B[1]), psi),
  P([0,0],[0,0])-expect(tensor(A[0], B[0]), psi),
  P([0,0],[0,1])-expect(tensor(A[0], B[1]), psi),
  P([0,0],[1,0])-expect(tensor(A[1], B[0]), psi),
  P([0,0],[1,1])-expect(tensor(A[1], B[1]), psi)]
```

We also have to define normalization of the subalgebras, in this case, only one:

```
behaviour_constraints.append("-0[0,0]+1.0")
```

From here, the solution follows the usual pathway:

```
level = 1
sdp = SdpRelaxation(P.get_all_operators(), normalized=False, verbose=1)
sdp.get_relaxation(level, objective=-P([0],[0],'A'),
                   momentequalities=behaviour_constraints,
                   substitutions=P.substitutions)
sdp.solve()
print(sdp.primal, sdp.dual)
```

Example 8: 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.

Then we set up the problem with specifically with the CHSH inequality in the probability picture as the objective function. This part is identical to the one discussed in Section [mixedlevel].

```
I = [[ 0,   -1,    0 ],
     [-1,    1,    1 ],
     [ 0,    1,   -1 ]]
P = Probability([2, 2], [2, 2])
objective = define_objective_with_I(I, P)
```

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:

```
level = 1
sdp = MoroderHierarchy([flatten(P.parties[0]), flatten(P.parties[1])],
                       verbose=1, normalized=False)
sdp.get_relaxation(level, objective=objective,
                   substitutions=P.substitutions)
```

We can further process the moment matrix, for instance, to impose partial positivity, or a matrix decomposition. To do these operations, we rely on PICOS:

```
Problem = sdp.convert_to_picos(duplicate_moment_matrix=True)
X = Problem.get_variable('X')
Y = Problem.get_variable('Y')
Z = Problem.add_variable('Z', (sdp.block_struct[0],
                         sdp.block_struct[0]))
Problem.add_constraint(Y.partial_transpose()>>0)
Problem.add_constraint(Z.partial_transpose()>>0)
Problem.add_constraint(X - Y + Z == 0)
Problem.add_constraint(Z[0,0] == 1)
solution = Problem.solve()
print(solution)
```

Alternatively, with SeDuMi's `fromsdpa` function (Sturm 1999), we can also impose the positivity of the partial trace of the moment matrix using MATLAB, or decompose the moment matrix in various forms. For this, we have to write the relaxation to a file:

```
sdp.write_to_file("chsh-moroder.dat-s")
```

If all we need is the partial positivity of the moment matrix, that is actually nothing but an extra symmetry. We can request this condition by passing an argument to the constructor, leading to a sparser SDP:

```
sdp = MoroderHierarchy([flatten(P.parties[0]), flatten(P.parties[1])],
                       verbose=1, ppt=True)
sdp.get_relaxation(level, objective=objective,
                   substitutions=P.substitutions)
```

CHAPTER 5

Revision History

Version 1.12.1

- Changed: Removed automated detection of simple moment substitutions.
- Changed: Better handling of monomial substitutions.

Version 1.12.0 (2016-11-28)

- New: Pass the optional *momentsubstitutions*= parameter to the *get_relaxation* method to substitute out specific moments.
- Changed: Warning message is displayed if the equality constraints are linearly dependent.
- Changed: CVXPY support improved, solver parameters passed on correctly. SCS can directly be requested as a solver.
- Fixed: Chordal graph extension works with blank objective functions and commuting variables.
- Fixed: Parallel computations produce weird deadlocks less frequently.

Version 1.11.1 (2016-08-07)

- Fixed: Major bug in generating localizing matrices with the correct monomials.
- Fixed: `fast_substitute` is able to handle some more extreme forms of commuting monomials.

Version 1.11.0 (2016-06-23)

- New: Experimental new parallel computation of the moment matrix and the constraints.
- New: CVXPY conversion with `convert_to_cvxpy`. CVXPY is now also a valid solver.
- New: The method `get_dual` returns the block in the dual solution corresponding to the requested constraint.
- Changed: Deprecated optional parameter `bounds` was removed.
- Fixed: Moments are correctly returned even if equalities are removed.
- Fixed: Constants in PICOS conversion are added correctly irrespective of where they are in the matrices.
- Fixed: PICOS conversion handles feasibility problems correctly.

- Fixed: The optional parameter `removeequalities=True` handles equalities of SDP variables correctly.

Version 1.10.3 (2016-02-26)

- Fixed: Problem with unexpanded moment equality constraints resolved.

Version 1.10.2 (2016-02-03)

- New: Very efficient substitutions of moment equalities if one side of the equality is the moment of a monomial, and the other side is a constant.

Version 1.10.1 (2016-01-29)

- Fixed: The moment equalities are removed correctly if asked.

Version 1.10 (2015-12-08)

- New: The function `generate_operators` returns a list of operators from the `sympy.physics.quantum` submodule. This is the old behaviour of `generate_variables`.
- New: The `SdpRelaxation` class is now subscriptable. You can retrieve the value of polynomials in the solved relaxation in such way. Internally, it calls `get_xmat_value` with `self`.
- New: The convenience method `solve()` was added to the class `SdpRelaxation`.
- New: The convenience method `write_to_file()` was added to the class `SdpRelaxation`.
- New: The convenience method `save_monomial_index()` was added to the class `SdpRelaxation`.
- New: The convenience method `find_solution_ranks()` was added to the class `SdpRelaxation`. It replaces the previous stand-alone `find_rank_loop()` function.
- New: The conversion routines `convert_to_picos` and `convert_to_mosek` are also part of the class `sdpRelaxation`.
- New: The new method `extract_dual_value()` was added to the class `SdpRelaxation` to calculate the inner product of the coefficient matrix of an SDP variable with the dual solution.
- New: The class `RdmHierarchy` was added to generate SDPs of the reduced density matrix method. Initial support for 1D spinless, translational invariant systems is included.
- New: Better support for the steering hierarchy in a new class `SteeringHierarchy`.
- Changed: The function `generate_variables` now returns a list of `sympy.Symbol` variables if commutative variables are requested, and the default is commutative.
- Changed: Many unnecessary user-facing functions were removed.
- Changed: The SOS decomposition is now requested with `get_sos_decomposition` from the class `SdpRelaxation`, and it returns a list of the SOS polynomials.
- Changed: The optional parameter `bounds` for `get_relaxation` is deprececated, use the optional parameters `momentinequalities` and `momentequalities` instead.
- Changed: Removed `convert_to_picos_extra_moment_matrix` and added optional parameter `duplicate_moment_matrix` to `convert_to_picos` to achieve the same effect.
- Changed: The chordal extension is now requested as an optional parameter `chordal_extension=True` passed to the `get_relaxation` method, and not by specifying it as a hierarchy type in the constructor.
- Changed: The Moroder hierarchy is now a class.
- Changed: Small improvements in speed in the substitution routines; unit tests for the substitution routines.
- Changed: The `read_sdpa_out` routine takes an optional argument for a relaxation, and adds the solution to this object if requested.

- Changed: Instead of an examples folder, all examples were migrated to the documentation.
- Changed: The symbolic variables which are not to be relaxed are now supplied to the constructor with the optional parameter `parameters`.
- Changed: Redundant positive-semidefinite constraint type removed.
- Fixed: PICOS and MOSEK conversion works for complex matrices too (issue #10).
- Fixed: The moment symmetries are correctly calculated for both Hermitian and non-Hermitian variables (issue #9)

Version 1.9 (2015-08-28)

- New: Defining the constraints now also allows using for the symbols <, <=, >=, >. Additionally, the function `Eq` from SymPy can be used to defined equalities.
- New: The function `solve_sdp` also accepts `solver="cvxopt"` to use CVXOPT for solving a relaxation (requires PICOS and CVXOPT).
- New: `convert_to_human_readable` function returns the objective function and the moment matrix as a string and a matrix of strings to give a symbolic representation of the problem.
- New: `get_next_neighbors` function retrieves the forward neighbors at a given distance of a site or set of sites in a lattice.
- New: Much faster substitutions if the right-hand side of the substitution never contains variables that are not in the left-hand side.
- New: Non-unique variables are considered only once in each variable set.
- New: When using `solve_sdp` to solve the relaxation, the solution, its status, and the time it takes to solve are now part of the class `SdpRelaxation`.
- New: The class `Probability` provides an intuitive way to define quantum probabilities and Bell inequalities.
- New: The function `solve_sdp` autodetects available solvers and complains if there is none.
- New: The optional parameter `solverparameters` to the function `solve_sdp` can contain a dictionary of options, with a different set for each of the target solvers.
- New: Regression testing framework added.
- Changed: The functions `find_rank_loop`, `sos_decomposition`, and `get_xmat_value` are no longer required an `x_mat` or `y_mat` parameter to pass the primal or dual solution. These values are extracted from the solved relaxation. The respective parameters became optional.
- Changed: Constant term in objective function is added to the primal and dual values when using the `solve_sdp` function.
- Changed: The primal and dual values of the Mosek solution change their signs when using the `solve_sdp` function.
- Changed: The verbosity parameter also controls the console output of every solver.
- Changed: Faacets relaxations got their own class `FaacetsRelaxation`.
- Fixed: Localizing matrices are built correctly when substitution rules contain polynomials and when the identity operator is not part of the monomial sets.
- Fixed: The function `get_xmat_value` also works in Pypy.

Version 1.8 (2015-05-25)

- New: Complex moment matrices are embedded to as real matrices in the SDPA export and the `solve_sdp` function.

- New: Localizing monomials can be fine-tuned by supplying them to `get_relaxation` through the optional parameter `localizing_monomials`.
- New: `solve_sdp` can also solve a problem with Mosek.
- New: The function `get_xmat_value` returns the matching value for a monomial from a solution matrix, given the relaxation and the solution.
- Changed: `solve_sdp` no longer accepts parameters `solutionmatrix` and `solverexecutable`. All parameters are now passed via the solverparameters dictionary.
- Changed: Legacy Picos code removed. Requirement is now Picos >=1.0.2.
- Fixed: Determining degree of polynomial also works when coefficient is complex.

Version 1.7 (2015-03-23)

- New: the function `find_rank_loop` aids the detection of a rank loop.
- New: the function `write_to_human_readable` writes the relaxation in a human-readable format.
- New: the function `read_sdpa_out` is now exposed to the user, primarily to help in detecting rank loops.
- New: the function `save_monomial_index` allows saving the monomial index of a relaxation.
- New: support for obtaining the SOS decomposition from a dual solution through the function `sos_decomposition`.
- New: optional parameter `psd=[matrix1, matrix2, ..., matrixn]` can be passed to `get_relaxation` and `process_constraints` which contain symbolic matrices that should be positive semidefinite.
- New: solution matrices can be returned by `solve_sdp` by passing the optional parameter `solutionmatrix=True`. It does not work for diagonal blocks.
- New: basic interface for Faacets via the function `get_faacets_relaxation`.
- New: PPT constraint can be imposed directly in the Moroder hierarchy by passing the extra parameter `ppt=True` to the constructor.
- New: Passing the optional parameter `extramomentmatrices=...` to `get_relaxation` allows defining new moment matrices either freely or based on the first one. Basic relations of the elements between the moment matrices can be imposed as strings passed through `inequalites=...`.
- Changed: Nieto-Silleras hierarchy is no longer supported through an option. Now constraints have to be manually defined.
- Changed: Monomials are not saved automatically with `verbose=2`.
- Fixed: wider range of substitutions supported, including a polynomial on the right-hands side of the substitution.
- Fixed: constraints for fermionic and bosonic systems and Pauli operators.

Version 1.6 (2014-12-22)

- Syntax for passing parameters changed. Only the level of the relaxation is compulsory for obtaining a relaxation.
- Extra parameter for bounds on the variables was added. Syntax is identical to the inequalities. The difference is that the inequalities in the bounds will not be relaxed by localizing matrices.
- Support for chordal graph extension in the commutative case (doi:10.1137/050623802). Pass `hierarchy="npa_chordal"` to the constructor.
- It is possible to pass variables which will not be relaxed. Pass `nonrelaxed=[variables]` to the constructor.

- It is possible to change the constraints once the moment matrix is generated. Refer to the new function `process_constraints`.
- Extra parameter `nsextraobjvars=[]` was added for passing additional variables to the Nieto-Silleras hierarchy. This is important because the top-left elements of the blocks of moment matrices in the relaxation are not one: they add up to one. Hence specifying the last element of a measurement becomes possible with this option. The number of elements in this must match the number of behaviours.
- PICOS conversion routines were separated and reworked to ensure sparsity.
- Moved documentation to Sphinx.
- SciPy dependency made optional.

Version 1.5 (2014-11-27)

- Support for Moroder hierarchy (doi:10.1103/PhysRevLett.111.030501).
- Further symmetries are discovered when all variables are Hermitian.
- Normalization can be turned off.

Version 1.4 (2014-11-18)

- Pypy support restored with limitations.
- Direct export to and optimization by MOSEK.
- Added helper function to add constraints on Pauli operators.
- Handling of complex coefficients improved.
- Added PICOS compatibility layer, enabling solving a problem by a larger range of solvers.
- Bug fixes: Python 3 compatibility restored.

Version 1.3 (2014-11-03)

- Much smaller SDPs are generated when using the helper functions for quantum correlations by not considering the last projector in the measurements and thus removing the sum-to-identity constraint; positive semidefinite condition is not influenced by this.
- Helper functions for fermionic systems and projective measurements are simplified.
- Support for the Nieto-Silleras (doi:10.1088/1367-2630/16/1/013035) hierarchy for level 1+ relaxations.

Version 1.2.4 (2014-06-13)

- Bug fixes: mixed commutative and noncommutative variable monomials are handled correctly in substitutions, constant integer objective functions are accepted.

Version 1.2.3 (2014-06-04)

- CHSH inequality added as an example.
- Allows supplying extra monomials to a given level of relaxation.
- Added functions to make it easier to work with Bell inequalities.
- Bug fixes: constant separation works correctly for integers, max-cut example fixed.

Version 1.2.2 (2014-05-27)

- Much faster SDPA writer for problems with many blocks.
- Removal of equalities does not happen by default.

Version 1.2.1 (2014-05-22)

- Size of localizing matrices adjusts to individual inequalities.
- Internal structure for storing monomials reorganized.
- Checks for maximum order in the constraints added.
- Fermionic constraints corrected.

Version 1.2 (2014-05-16)

- Fast replace was updated and made default.
- Numpy and SciPy are now dependencies.
- Replaced internal data structures by SciPy sparse matrices.
- Pypy is no longer supported.
- Equality constraints are removed by a QR decomposition and basis transformation.
- Functions added to support calling SDPA from Python.
- Helper functions added to help phrasing physics problems.
- More commutative examples added for comparison to Gloptipoly.
- Internal module structure reorganized.

Version 1.1 (2014-05-12)

- Commutative variables also work.
- Major rework of how the moment matrix is generated.

Version 1.0 (2014-04-29)

- Initial release.

CHAPTER 6

Function Reference

SdpRelaxation Class

class ncpol2sdpa.**SdpRelaxation**(*variables*, *parameters=None*, *verbose=0*, *normalized=True*, *parallel=False*)

Class for obtaining sparse SDP relaxation.

Parameters

- **variables** (list of sympy.physics.quantum.operator.Operator or sympy.physics.quantum.operator.HermitianOperator or a list of list.) – Commutative or noncommutative, Hermitian or nonhermiatian variables, possibly a list of list of variables if the hierarchy is not NPA.
- **parameters** (list of sympy.physics.quantum.operator.Operator or sympy.physics.quantum.operator.HermitianOperator or a list of list.) – Optional symbolic variables for which moments are not generated.
- **verbose** (*int.*) – Optional parameter for level of verbosity:
  - 0: quiet (default)
  - 1: verbose
  - 2: debug level
- **normalized** (*bool.*) – Optional parameter for changing the normalization of states over which the optimization happens. Turn it off if further processing is done on the SDP matrix before solving it.
- **parallel** (*bool.*) – Optional parameter for allowing parallel computations.

Attributes:

- *monomial_sets*: The monomial sets that generate the moment matrix blocks.
- *monomial_index*: Dictionary that maps monomials to SDP variables.

- *constraints*: The complete set of constraints after preprocesssing.
- *primal*: The primal optimal value.
- *dual*: The dual optimal value.
- *x_mat*: The primal solution matrix.
- *y_mat*: The dual solution matrix.
- *solution_time*: The amount of time taken to solve the relaxation.
- *status*: The solution status of the relaxation.

__getitem__(*index*)
Obtained the value for a polynomial in a solved relaxation.

Parameters **index** (*sympy.core.exp.Expr*) – The polynomial.

Returns The value of the polynomial extracted from the solved SDP.

Return type float

convert_to_mosek()
Convert an SDP relaxation to a MOSEK task.

Returns `mosek.Task`.

convert_to_picos(*duplicate_moment_matrix=False*)
Convert the SDP relaxation to a PICOS problem such that the exported .dat-s file is extremely sparse, there is not penalty imposed in terms of SDP variables or number of constraints. This conversion can be used for imposing extra constraints on the moment matrix, such as partial transpose.

Parameters **duplicate_moment_matrix** (*bool.*) – Optional parameter to add an unconstrained moment matrix to the problem with the same structure as the moment matrix with the PSD constraint.

Returns `picos.Problem`.

extract_dual_value(*monomial*, *blocks=None*)
Given a solution of the dual problem and a monomial, it returns the inner product of the corresponding coefficient matrix and the dual solution. It can be restricted to certain blocks.

Parameters

- **monomial** (`sympy.core.exp.Expr`.) – The monomial for which the value is requested.
- **monomial** – The monomial for which the value is requested.
- **blocks** (list of *int*.) – Optional parameter to specify the blocks to be included.

Returns The value of the monomial in the solved relaxation.

Return type float.

find_solution_ranks(*xmat=None*, *baselevel=0*)
Helper function to detect rank loop in the solution matrix.

Parameters

- **sdpRelaxation** (*ncpol2sdpa.SdpRelaxation*.) – The SDP relaxation.
- **x_mat** (`numpy.array`.) – Optional parameter providing the primal solution of the moment matrix. If not provided, the solution is extracted from the sdpRelaxation object.

- **base_level** (*int.*) – Optional parameter for specifying the lower level relaxation for which the rank loop should be tested against.

Returns list of int – the ranks of the solution matrix with in the order of increasing degree.

get_relaxation(*level*, *objective=None*, *inequalities=None*, *equalities=None*, *substitutions=None*, *momentinequalities=None*, *momentequalities=None*, *momentsubstitutions=None*, *removeequalities=False*, *extramonomials=None*, *extramomentmatrices=None*, *extraobjexpr=None*, *localizing_monomials=None*, *chordal_extension=False*)

Get the SDP relaxation of a noncommutative polynomial optimization problem.

Parameters

- **level** (*int.*) – The level of the relaxation. The value -1 will skip automatic monomial generation and use only the monomials supplied by the option *extramonomials*.
- **obj** (`sympy.core.exp.Expr`.) – Optional parameter to describe the objective function.
- **inequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list inequality constraints.
- **equalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list equality constraints.
- **substitutions** (dict of `sympy.core.exp.Expr`.) – Optional parameter containing monomials that can be replaced (e.g., idempotent variables).
- **momentinequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of inequalities defined on moments.
- **momentequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of equalities defined on moments.
- **momentsubstitutions** (dict of `sympy.core.exp.Expr`.) – Optional parameter containing moments that can be replaced.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **extramonomials** (list of `sympy.core.exp.Expr`.) – Optional paramter of monomials to be included, on top of the requested level of relaxation.
- **extramomentmatrices** (*list of list of str.*) – Optional paramter of duplicating or adding moment matrices. A new moment matrix can be unconstrained (""), a copy of the first one ("copy"), and satisfying a partial positivity constraint ("ppt"). Each new moment matrix is requested as a list of string of these options. For instance, adding a single new moment matrix as a copy of the first would be `extramomentmatrices=[["copy"]]`.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function.
- **localizing_monomials** (list of list of *sympy.core.exp.Expr*.) – Optional parameter to specify sets of localizing monomials for each constraint. The internal order of constraints is inequalities first, followed by the equalities. If the parameter is specified, but for a certain constraint the automatic localization is requested, leave None in its place in this parameter.
- **chordal_extension** (*bool.*) – Optional parameter to request a sparse chordal extension.

get_sos_decomposition(*threshold=0.0*)
Given a solution of the dual problem, it returns the SOS decomposition.

Parameters **threshold** (*float.*) – Optional parameter for specifying the threshold value below which the eigenvalues and entries of the eigenvectors are disregarded.

Returns The SOS decomposition of [sigma_0, sigma_1, ..., sigma_m]

Return type list of `sympy.core.exp.Expr`.

process_constraints(*inequalities=None*, *equalities=None*, *momentinequalities=None*, *momenteequalities=None*, *block_index=0*, *removeequalities=False*)
Process the constraints and generate localizing matrices. Useful only if the moment matrix already exists. Call it if you want to replace your constraints. The number of the respective types of constraints and the maximum degree of each constraint must remain the same.

Parameters

- **inequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list inequality constraints.
- **equalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list equality constraints.
- **momentinequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of inequalities defined on moments.
- **momentequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of equalities defined on moments.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **removeequalities** – Optional parameter to attempt removing the equalities by solving the linear equations.

save_monomial_index(*filename*)
Write the monomial index to a file.

Parameters **filename** (*str.*) – The name of the file to write to.

set_objective(*objective*, *extraobjexpr=None*)
Set or change the objective function of the polynomial optimization problem.

Parameters

- **objective** (`sympy.core.expr.Expr`) – Describes the objective function.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function

solve(*solver=None*, *solverparameters=None*)
Call a solver on the SDP relaxation. Upon successful solution, it returns the primal and dual objective values along with the solution matrices. It also sets these values in the *sdpRelaxation* object, along with some status information.

Parameters

- **sdpRelaxation** (*ncpol2sdpa.SdpRelaxation.*) – The SDP relaxation to be solved.
- **solver** (*str.*) – The solver to be called, either *None*, "sdpa", "mosek", "cvxpy", "scs", or "cvxopt". The default is *None*, which triggers autodetect.

- **solverparameters** (*dict of str.*) – Parameters to be passed to the solver. Actual options depend on the solver:

  SDPA:

  - *"executable"*: Specify the executable for SDPA. E.g., *"executable":"/usr/local/bin/sdpa"*, or *"executable":"sdpa_gmp"*
  - *"paramsfile"*: Specify the parameter file

  Mosek: Refer to the Mosek documentation. All arguments are passed on.

  Cvxopt: Refer to the PICOS documentation. All arguments are passed on.

  Cvxpy: Refer to the Cvxpy documentation. All arguments are passed on.

  SCS: Refer to the Cvxpy documentation. All arguments are passed on.

write_to_file(*filename*, *filetype=None*)

Write the relaxation to a file.

Parameters

- **filename** (*str.*) – The name of the file to write to. The type can be autodetected from the extension: .dat-s for SDPA, .task for mosek or .csv for human readable format.
- **filetype** (*str.*) – Optional parameter to define the filetype. It can be "sdpa" for SDPA , "mosek" for Mosek, or "csv" for human readable format.

MoroderHierarchy Class

class ncpol2sdpa.**SteeringHierarchy**(*variables*, *verbose=0*, *matrix_var_dim=None*, *mark_conjugate=False*, *parallel=False*)

Class for obtaining a step in the steering hierarchy.

Parameters

- **variables** (list of sympy.physics.quantum.operator.Operator or sympy.physics.quantum.operator.HermitianOperator or a list of list.) – Commutative or noncommutative, Hermitian or nonhermiatian variables.
- **verbose** (*int.*) – Optional parameter for level of verbosity:
  - 0: quiet
  - 1: verbose
  - 2: debug level
- **matrix_var_dim** (*int.*) – Optional parameter to specify the size of matrix variable blocks
- **mark_conjugate** (*bool.*) – Use this optional parameter to generate a symbolic representation of the steering hierarchy for export.

Attributes:

- *monomial_sets*: The monomial sets that generate the moment matrix blocks.
- *monomial_index*: Dictionary that maps monomials to SDP variables.
- *constraints*: The complete set of constraints after preprocesssing.
- *primal*: The primal optimal value.

- *dual*: The dual optimal value.
- *x_mat*: The primal solution matrix.
- *y_mat*: The dual solution matrix.
- *solution_time*: The amount of time taken to solve the relaxation.
- *status*: The solution status of the relaxation.

__getitem__(*index*)
Obtained the value for a polynomial in a solved relaxation.

Parameters **index** (*sympy.core.exp.Expr*) – The polynomial.

Returns The value of the polynomial extracted from the solved SDP.

Return type float

convert_to_mosek()
Convert an SDP relaxation to a MOSEK task.

Returns `mosek.Task`.

convert_to_picos(*duplicate_moment_matrix=False*)
Convert the SDP relaxation to a PICOS problem such that the exported .dat-s file is extremely sparse, there is not penalty imposed in terms of SDP variables or number of constraints. This conversion can be used for imposing extra constraints on the moment matrix, such as partial transpose.

Parameters **duplicate_moment_matrix** (*bool.*) – Optional parameter to add an unconstrained moment matrix to the problem with the same structure as the moment matrix with the PSD constraint.

Returns `picos.Problem`.

get_relaxation(*level*, *objective=None*, *inequalities=None*, *equalities=None*, *substitutions=None*, *momentinequalities=None*, *momentequalities=None*, *momentsubstitutions=None*, *removeequalities=False*, *extramonomials=None*, *extramomentmatrices=None*, *extraobjexpr=None*, *localizing_monomials=None*, *chordal_extension=False*)
Get the SDP relaxation of a noncommutative polynomial optimization problem.

Parameters

- **level** (*int.*) – The level of the relaxation. The value -1 will skip automatic monomial generation and use only the monomials supplied by the option *extramonomials*.
- **obj** (`sympy.core.exp.Expr`.) – Optional parameter to describe the objective function.
- **inequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list inequality constraints.
- **equalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list equality constraints.
- **substitutions** (dict of `sympy.core.exp.Expr`.) – Optional parameter containing monomials that can be replaced (e.g., idempotent variables).
- **momentinequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of inequalities defined on moments.
- **momentequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of equalities defined on moments.

- **momentsubstitutions** (dict of `sympy.core.exp.Expr`.) – Optional parameter containing moments that can be replaced.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **extramonomials** (list of `sympy.core.exp.Expr`.) – Optional paramter of monomials to be included, on top of the requested level of relaxation.
- **extramomentmatrices** (*list of list of str.*) – Optional paramter of duplicating or adding moment matrices. A new moment matrix can be unconstrained (""), a copy of the first one ("copy"), and satisfying a partial positivity constraint ("ppt"). Each new moment matrix is requested as a list of string of these options. For instance, adding a single new moment matrix as a copy of the first would be `extramomentmatrices=[["copy"]]`.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function.
- **localizing_monomials** (list of list of *sympy.core.exp.Expr*.) – Optional parameter to specify sets of localizing monomials for each constraint. The internal order of constraints is inequalities first, followed by the equalities. If the parameter is specified, but for a certain constraint the automatic localization is requested, leave None in its place in this parameter.
- **chordal_extension** (*bool.*) – Optional parameter to request a sparse chordal extension.

process_constraints(*inequalities=None*, *equalities=None*, *momentinequalities=None*, *momentequalities=None*, *block_index=0*, *removeequalities=False*)

Process the constraints and generate localizing matrices. Useful only if the moment matrix already exists. Call it if you want to replace your constraints. The number of the respective types of constraints and the maximum degree of each constraint must remain the same.

Parameters

- **inequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list inequality constraints.
- **equalities** (list of `sympy.core.exp.Expr`.) – Optional parameter to list equality constraints.
- **momentinequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of inequalities defined on moments.
- **momentequalities** (list of `sympy.core.exp.Expr`.) – Optional parameter of equalities defined on moments.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **removeequalities** – Optional parameter to attempt removing the equalities by solving the linear equations.

save_monomial_index(*filename*)

Write the monomial index to a file.

Parameters **filename** (*str.*) – The name of the file to write to.

set_objective(*objective*, *extraobjexpr=None*)

Set or change the objective function of the polynomial optimization problem.

Parameters

- **objective** (sympy.core.expr.Expr) – Describes the objective function.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function

solve(*solver=None*, *solverparameters=None*)

Call a solver on the SDP relaxation. Upon successful solution, it returns the primal and dual objective values along with the solution matrices. It also sets these values in the *sdpRelaxation* object, along with some status information.

Parameters

- **sdpRelaxation** (*ncpol2sdpa.SdpRelaxation.*) – The SDP relaxation to be solved.
- **solver** (*str.*) – The solver to be called, either *None*, "sdpa", "mosek", "cvxpy", "scs", or "cvxopt". The default is *None*, which triggers autodetect.
- **solverparameters** (*dict of str.*) – Parameters to be passed to the solver. Actual options depend on the solver:

  SDPA:

  – *"executable"*: Specify the executable for SDPA. E.g., *"executable":"/usr/local/bin/sdpa"*, or *"executable":"sdpa_gmp"*

  – *"paramsfile"*: Specify the parameter file

  Mosek: Refer to the Mosek documentation. All arguments are passed on.

  Cvxopt: Refer to the PICOS documentation. All arguments are passed on.

  Cvxpy: Refer to the Cvxpy documentation. All arguments are passed on.

  SCS: Refer to the Cvxpy documentation. All arguments are passed on.

write_to_file(*filename*, *filetype=None*)

Write the relaxation to a file.

Parameters

- **filename** (*str.*) – The name of the file to write to. The type can be autodetected from the extension: .dat-s for SDPA, .task for mosek, .csv for human readable format, or .txt for a symbolic export
- **filetype** (*str.*) – Optional parameter to define the filetype. It can be "sdpa" for SDPA , "mosek" for Mosek, "csv" for human readable format, or "txt" for a symbolic export.

SteeringHierarchy Class

class ncpol2sdpa.**SteeringHierarchy**(*variables*, *verbose=0*, *matrix_var_dim=None*, *mark_conjugate=False*, *parallel=False*)

Class for obtaining a step in the steering hierarchy.

Parameters

- **variables** (list of sympy.physics.quantum.operator.Operator or sympy.physics.quantum.operator.HermitianOperator or a list of list.) – Commutative or noncommutative, Hermitian or nonhermiatian variables.
- **verbose** (*int.*) – Optional parameter for level of verbosity:

  – 0: quiet

  - 1: verbose
  - 2: debug level
- **matrix_var_dim** (*int.*) – Optional parameter to specify the size of matrix variable blocks
- **mark_conjugate** (*bool.*) – Use this optional parameter to generate a symbolic representation of the steering hierarchy for export.

Attributes:

- *monomial_sets*: The monomial sets that generate the moment matrix blocks.
- *monomial_index*: Dictionary that maps monomials to SDP variables.
- *constraints*: The complete set of constraints after preprocesssing.
- *primal*: The primal optimal value.
- *dual*: The dual optimal value.
- *x_mat*: The primal solution matrix.
- *y_mat*: The dual solution matrix.
- *solution_time*: The amount of time taken to solve the relaxation.
- *status*: The solution status of the relaxation.

__getitem__(*index*)

Obtained the value for a polynomial in a solved relaxation.

Parameters **index** (*sympy.core.exp.Expr*) – The polynomial.

Returns The value of the polynomial extracted from the solved SDP.

Return type float

convert_to_mosek()

Convert an SDP relaxation to a MOSEK task.

Returns `mosek.Task`.

convert_to_picos(*duplicate_moment_matrix=False*)

Convert the SDP relaxation to a PICOS problem such that the exported .dat-s file is extremely sparse, there is not penalty imposed in terms of SDP variables or number of constraints. This conversion can be used for imposing extra constraints on the moment matrix, such as partial transpose.

Parameters **duplicate_moment_matrix** (*bool.*) – Optional parameter to add an unconstrained moment matrix to the problem with the same structure as the moment matrix with the PSD constraint.

Returns `picos.Problem`.

get_relaxation(*level*, *objective=None*, *inequalities=None*, *equalities=None*, *substitutions=None*, *momentinequalities=None*, *momentequalities=None*, *momentsubstitutions=None*, *removeequalities=False*, *extramonomials=None*, *extramomentmatrices=None*, *extraobjexpr=None*, *localizing_monomials=None*, *chordal_extension=False*)

Get the SDP relaxation of a noncommutative polynomial optimization problem.

Parameters

- **level** (*int.*) – The level of the relaxation. The value -1 will skip automatic monomial generation and use only the monomials supplied by the option *extramonomials*.

- **obj** (sympy.core.exp.Expr.) – Optional parameter to describe the objective function.
- **inequalities** (list of sympy.core.exp.Expr.) – Optional parameter to list inequality constraints.
- **equalities** (list of sympy.core.exp.Expr.) – Optional parameter to list equality constraints.
- **substitutions** (dict of sympy.core.exp.Expr.) – Optional parameter containing monomials that can be replaced (e.g., idempotent variables).
- **momentinequalities** (list of sympy.core.exp.Expr.) – Optional parameter of inequalities defined on moments.
- **momentequalities** (list of sympy.core.exp.Expr.) – Optional parameter of equalities defined on moments.
- **momentsubstitutions** (dict of sympy.core.exp.Expr.) – Optional parameter containing moments that can be replaced.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **extramonomials** (list of sympy.core.exp.Expr.) – Optional paramter of monomials to be included, on top of the requested level of relaxation.
- **extramomentmatrices** (*list of list of str.*) – Optional paramter of duplicating or adding moment matrices. A new moment matrix can be unconstrained (""), a copy of the first one ("copy"), and satisfying a partial positivity constraint ("ppt"). Each new moment matrix is requested as a list of string of these options. For instance, adding a single new moment matrix as a copy of the first would be `extramomentmatrices=[["copy"]]`.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function.
- **localizing_monomials** (list of list of *sympy.core.exp.Expr.*) – Optional parameter to specify sets of localizing monomials for each constraint. The internal order of constraints is inequalities first, followed by the equalities. If the parameter is specified, but for a certain constraint the automatic localization is requested, leave None in its place in this parameter.
- **chordal_extension** (*bool.*) – Optional parameter to request a sparse chordal extension.

process_constraints(*inequalities=None*, *equalities=None*, *momentinequalities=None*, *momentequalities=None*, *block_index=0*, *removeequalities=False*)

Process the constraints and generate localizing matrices. Useful only if the moment matrix already exists. Call it if you want to replace your constraints. The number of the respective types of constraints and the maximum degree of each constraint must remain the same.

Parameters

- **inequalities** (list of sympy.core.exp.Expr.) – Optional parameter to list inequality constraints.
- **equalities** (list of sympy.core.exp.Expr.) – Optional parameter to list equality constraints.
- **momentinequalities** (list of sympy.core.exp.Expr.) – Optional parameter of inequalities defined on moments.

- **momentequalities** (list of sympy.core.exp.Expr.) – Optional parameter of equalities defined on moments.
- **removeequalities** (*bool.*) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **removeequalities** – Optional parameter to attempt removing the equalities by solving the linear equations.

save_monomial_index(*filename*)
Write the monomial index to a file.

Parameters **filename** (*str.*) – The name of the file to write to.

set_objective(*objective*, *extraobjexpr=None*)
Set or change the objective function of the polynomial optimization problem.

Parameters

- **objective** (sympy.core.expr.Expr) – Describes the objective function.
- **extraobjexpr** (*str.*) – Optional parameter of a string expression of a linear combination of moment matrix elements to be included in the objective function

solve(*solver=None*, *solverparameters=None*)
Call a solver on the SDP relaxation. Upon successful solution, it returns the primal and dual objective values along with the solution matrices. It also sets these values in the *sdpRelaxation* object, along with some status information.

Parameters

- **sdpRelaxation** (*ncpol2sdpa.SdpRelaxation.*) – The SDP relaxation to be solved.
- **solver** (*str.*) – The solver to be called, either *None*, "sdpa", "mosek", "cvxpy", "scs", or "cvxopt". The default is *None*, which triggers autodetect.
- **solverparameters** (*dict of str.*) – Parameters to be passed to the solver. Actual options depend on the solver:

  SDPA:

  – *"executable"*: Specify the executable for SDPA. E.g., *"executable":"/usr/local/bin/sdpa"*, or *"executable":"sdpa_gmp"*

  – *"paramsfile"*: Specify the parameter file

  Mosek: Refer to the Mosek documentation. All arguments are passed on.

  Cvxopt: Refer to the PICOS documentation. All arguments are passed on.

  Cvxpy: Refer to the Cvxpy documentation. All arguments are passed on.

  SCS: Refer to the Cvxpy documentation. All arguments are passed on.

write_to_file(*filename*, *filetype=None*)
Write the relaxation to a file.

Parameters

- **filename** (*str.*) – The name of the file to write to. The type can be autodetected from the extension: .dat-s for SDPA, .task for mosek, .csv for human readable format, or .txt for a symbolic export
- **filetype** (*str.*) – Optional parameter to define the filetype. It can be "sdpa" for SDPA , "mosek" for Mosek, "csv" for human readable format, or "txt" for a symbolic export.

FaacetsRelaxation Class

class ncpol2sdpa.**FaacetsRelaxation**
Class for wrapping around a Faacets relaxation.

get_relaxation(*A_configuration*, *B_configuration*, *I*)
Get the sparse SDP relaxation of a Bell inequality.

Parameters

- **A_configuration** (*list of list of int.*) – The definition of measurements of Alice.
- **B_configuration** (*list of list of int.*) – The definition of measurements of Bob.
- **I** (*list of list of int.*) – The matrix describing the Bell inequality in the Collins-Gisin picture.

solve(*solver=None*, *solverparameters=None*)
Call a solver on the SDP relaxation. Upon successful solution, it returns the primal and dual objective values along with the solution matrices. It also sets these values in the *sdpRelaxation* object, along with some status information.

Parameters

- **sdpRelaxation** (*ncpol2sdpa.SdpRelaxation.*) – The SDP relaxation to be solved.
- **solver** (*str.*) – The solver to be called, either *None*, "sdpa", "mosek", "cvxpy", "scs", or "cvxopt". The default is *None*, which triggers autodetect.
- **solverparameters** (*dict of str.*) – Parameters to be passed to the solver. Actual options depend on the solver:

  SDPA:

  – *"executable"*: Specify the executable for SDPA. E.g., *"executable":"/usr/local/bin/sdpa"*, or *"executable":"sdpa_gmp"*

  – *"paramsfile"*: Specify the parameter file

  Mosek: Refer to the Mosek documentation. All arguments are passed on.

  Cvxopt: Refer to the PICOS documentation. All arguments are passed on.

  Cvxpy: Refer to the Cvxpy documentation. All arguments are passed on.

  SCS: Refer to the Cvxpy documentation. All arguments are passed on.

Functions to Help Define Polynomial Optimization Problems

ncpol2sdpa.**generate_operators**(*name*, *n_vars=1*, *hermitian=None*, *commutative=False*)
Generates a number of commutative or noncommutative operators

Parameters

- **name** (*str.*) – The prefix in the symbolic representation of the noncommuting variables. This will be suffixed by a number from 0 to n_vars-1 if n_vars > 1.
- **n_vars** (*int.*) – The number of variables.

- **hermitian** (*bool.*) – Optional parameter to request Hermitian variables .
- **commutative** (*bool.*) – Optional parameter to request commutative variables. Commutative variables are Hermitian by default.

Returns list of `sympy.physics.quantum.operator.Operator` or `sympy.physics.quantum.operator.HermitianOperator` variables

Example

```
>>> generate_variables('y', 2, commutative=True)
[y0, y1]
```

ncpol2sdpa.**generate_variables**(*name*, *n_vars=1*, *hermitian=None*, *commutative=True*)

Generates a number of commutative or noncommutative variables

Parameters

- **name** (*str.*) – The prefix in the symbolic representation of the noncommuting variables. This will be suffixed by a number from 0 to n_vars-1 if n_vars > 1.
- **n_vars** (*int.*) – The number of variables.
- **hermitian** (*bool.*) – Optional parameter to request Hermitian variables .
- **commutative** (*bool.*) – Optional parameter to request commutative variables. Commutative variables are Hermitian by default.

Returns list of `sympy.physics.quantum.operator.Operator` or `sympy.physics.quantum.operator.HermitianOperator` variables or *sympy.Symbol*

Example

```
>>> generate_variables('y', 2, commutative=True)
[y0, y1]
```

ncpol2sdpa.**get_monomials**(*variables*, *degree*)

Generates all noncommutative monomials up to a degree

Parameters

- **variables** (list of `sympy.physics.quantum.operator.Operator` or `sympy.physics.quantum.operator.HermitianOperator`.) – The noncommutative variables to generate monomials from
- **degree** (*int.*) – The maximum degree.

Returns list of monomials.

ncpol2sdpa.**flatten**(*lol*)

Flatten a list of lists to a list.

Parameters **lol** (*list of list.*) – A list of lists in arbitrary depth.

Returns flat list of elements.

Functions to Study Output of Solver

ncpol2sdpa.**read_sdpa_out**(*filename*, *solutionmatrix=False*, *status=False*, *sdp=None*)

Helper function to parse the output file of SDPA.

Parameters

- **filename** (*str.*) – The name of the SDPA output file.
- **solutionmatrix** (*bool.*) – Optional parameter for retrieving the solution.
- **status** (*bool.*) – Optional parameter for retrieving the status.
- **sdp** (*sdp.*) – Optional parameter to add the solution to a relaxation.

Returns tuple of two floats and optionally two lists of *numpy.array* and a status string

Functions and Classes to Define Physics Problems

class ncpol2sdpa.**Probability**(**args*, ***kwargs*)

__call__(*output_*, *input_*, *marginal=None*)

Obtain your probabilities in the p(ab...|xy...) notation.

Parameters

- **output** (*list of ints.*) – Conditional output as [a, b, ...]
- **input** (*list of ints.*) – The input to condition on as [x, y, ...]
- **marginal** (*list of str.*) – Optional parameter. If it is a marginal, then you can define which party or parties it belongs to.

Returns polynomial of *sympy.physics.quantum.HermitianOperator*.

Example

For the CHSH scenario, to get p(10|01), write

P([1,0], [0,1])

To get the marginal p_A(0|1), write

P([0], [1], ['A'])

get_all_operators()

Return all operators across all parties and measurements to supply them to the *ncpol2sdpa.SdpRelaxation* class.

ncpol2sdpa.**bosonic_constraints**(*a*)

Return a set of constraints that define fermionic ladder operators.

Parameters **a** (list of sympy.physics.quantum.operator.Operator.) – The non-Hermitian variables.

Returns a dict of substitutions.

ncpol2sdpa.**fermionic_constraints**(*a*)

Return a set of constraints that define fermionic ladder operators.

Parameters **a** (list of sympy.physics.quantum.operator.Operator.) – The non-Hermitian variables.

Returns a dict of substitutions.

ncpol2sdpa.**pauli_constraints**(*X*, *Y*, *Z*)

Return a set of constraints that define Pauli spin operators.

Parameters

- **X** (list of `sympy.physics.quantum.operator.HermitianOperator`.) – List of Pauli X operator on sites.
- **Y** (list of `sympy.physics.quantum.operator.HermitianOperator`.) – List of Pauli Y operator on sites.
- **Z** (list of `sympy.physics.quantum.operator.HermitianOperator`.) – List of Pauli Z operator on sites.

Returns tuple of substitutions and equalities.

`ncpol2sdpa.`**`get_neighbors`**(*index*, *lattice_length*, *width=0*, *periodic=False*)
Get the forward neighbors of a site in a lattice.

Parameters

- **index** (*int.*) – Linear index of operator.
- **lattice_length** (*int.*) – The size of the 2D lattice in either dimension
- **width** (*int.*) – Optional parameter to define width.
- **periodic** (*bool*) – Optional parameter to indicate periodic boundary conditions.

Returns list of int – the neighbors in linear index.

`ncpol2sdpa.`**`get_next_neighbors`**(*indices*, *lattice_length*, *width=0*, *distance=1*, *periodic=False*)
Get the forward neighbors at a given distance of a site or set of sites in a lattice.

Parameters

- **index** (*int.*) – Linear index of operator.
- **lattice_length** (*int.*) – The size of the 2D lattice in either dimension
- **width** (*int.*) – Optional parameter to define width.
- **distance** – Optional parameter to define distance.
- **periodic** (*bool*) – Optional parameter to indicate periodic boundary conditions.

Returns list of int – the neighbors at given distance in linear index.

`ncpol2sdpa.`**`correlator`**(*A*, *B*)
Correlators between the probabilities of two parties.

Parameters

- **A** (list of list of `sympy.physics.quantum.operator.HermitianOperator`.) – Measurements of Alice.
- **B** (list of list of `sympy.physics.quantum.operator.HermitianOperator`.) – Measurements of Bob.

Returns list of correlators.

`ncpol2sdpa.`**`generate_measurements`**(*party*, *label*)
Generate variables that behave like measurements.

Parameters

- **party** (*list of int.*) – The list of number of measurement outputs a party has.
- **label** (*str.*) – The label to be given to the symbolic variables.

Returns list of list of `sympy.physics.quantum.operator.HermitianOperator`.

ncpol2sdpa.**projective_measurement_constraints**(**parties*)
Return a set of constraints that define projective measurements.

Parameters **parties** – Measurements of different parties.

Returns substitutions containing idempotency, orthogonality and commutation relations.

ncpol2sdpa.**maximum_violation**(*A_configuration*, *B_configuration*, *I*, *level*, *extra=None*)
Get the maximum violation of a two-party Bell inequality.

Parameters

- **A_configuration** (*list of int.*) – Measurement settings of Alice.
- **B_configuration** (*list of int.*) – Measurement settings of Bob.
- **I** (*list of list of int.*) – The I matrix of a Bell inequality in the Collins-Gisin notation.
- **level** (*int.*) – Level of relaxation.

Returns tuple of primal and dual solutions of the SDP relaxation.

ncpol2sdpa.**define_objective_with_I**(*I*, **args*)
Define a polynomial using measurements and an I matrix describing a Bell inequality.

Parameters

- **I** (*list of list of int.*) – The I matrix of a Bell inequality in the Collins-Gisin notation.
- **args** – Either the measurements of Alice and Bob or a *Probability* class describing their measurement operators.

Returns sympy.core.expr.Expr – the objective function to be solved as a minimization problem to find the maximum quantum violation. Note that the sign is flipped compared to the Bell inequality.

CHAPTER 7

References

Bancal, J.D., L. Sheridan, and V. Scarani. 2014. "More Randomness from the Same Data." *New Journal of Physics* 16 (3): 033011. doi:10.1088/1367-2630/16/3/033011.

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

Henrion, D., J. Lasserre and J. Löfberg. 2009. "GloptiPoly 3: Moments, Optimization and Semidefinite Programming." *Optimization Methods & Software*, 24: 761–779.

Johansson, J.R., P.D. Nation, and Franco Nori. 2013. "QuTiP 2: A Python Framework for the Dynamics of Open Quantum Systems." *Computer Physics Communications* 184 (4): 1234–40. doi:10.1016/j.cpc.2012.11.019.

Lasserre, J. B. 2010. "A Joint+Marginal Approach to Parametric Polynomial Optimization." *SIAM Journal on Optimization* 20(4):1995–2022. doi:10.1137/090759240.

Moroder, Tobias, Jean-Daniel Bancal, Yeong-Cherng Liang, Martin Hofmann, and Otfried Gühne. 2013. "Device-Independent Entanglement Quantification and Related Applications." *Physics Review Letters* 111 (3). American Physical Society: 030501. doi:10.1103/PhysRevLett.111.030501.

Navascués, M., A. García-Sáez, A. Acín, S. Pironio, and M.B. Plenio. 2013. "A Paradox in Bosonic Energy Computations via Semidefinite Programming Relaxations." *New Journal of Physics* 15 (2): 023026. doi:10.1088/1367-2630/15/2/023026.

Nieto-Silleras, O., S. Pironio, and J. Silman. 2014. "Using Complete Measurement Statistics for Optimal Device-Independent Randomness Evaluation." *New Journal of Physics* 16 (1): 013035. doi:10.1088/1367-2630/16/1/013035.

Pironio, S., M. Navascués, and A. Acín. 2010. "Convergent Relaxations of Polynomial Optimization Problems with Noncommuting Variables." *SIAM Journal on Optimization* 20 (5). SIAM: 2157–80. doi:10.1137/090760155.

Sturm, J.F. 1999. "Using SeDuMi 1.02, a MATLAB Toolbox for Optimization over Symmetric Cones." *Optimization Methods and Software* 11 (1-4): 625–53.

Waki, H.; S. Kim, M. Kojima, M. Muramatsu, and H. Sugimoto. 2008. "Algorithm 883: SparsePOP—A Sparse Semidefinite Programming Relaxation of Polynomial Optimization Problems." ACM Transactions on Mathematical Software, 2008, 35(2), 15. doi:10.1145/1377612.1377619.

Yamashita, M., K. Fujisawa, and M. Kojima. 2003. "SDPARA: Semidefinite Programming Algorithm Parallel Version." *Parallel Computing* 29 (8): 1053–67.

Index

Symbols

__call__() (ncpol2sdpa.Probability method), 36
__getitem__() (ncpol2sdpa.SdpRelaxation method), 24
__getitem__() (ncpol2sdpa.SteeringHierarchy method), 28, 31

B

bosonic_constraints() (in module ncpol2sdpa), 36

C

convert_to_mosek() (ncpol2sdpa.SdpRelaxation method), 24
convert_to_mosek() (ncpol2sdpa.SteeringHierarchy method), 28, 31
convert_to_picos() (ncpol2sdpa.SdpRelaxation method), 24
convert_to_picos() (ncpol2sdpa.SteeringHierarchy method), 28, 31
correlator() (in module ncpol2sdpa), 37

D

define_objective_with_I() (in module ncpol2sdpa), 38

E

extract_dual_value() (ncpol2sdpa.SdpRelaxation method), 24

F

FaacetsRelaxation (class in ncpol2sdpa), 34
fermionic_constraints() (in module ncpol2sdpa), 36
find_solution_ranks() (ncpol2sdpa.SdpRelaxation method), 24
flatten() (in module ncpol2sdpa), 35

G

generate_measurements() (in module ncpol2sdpa), 37
generate_operators() (in module ncpol2sdpa), 34
generate_variables() (in module ncpol2sdpa), 35
get_all_operators() (ncpol2sdpa.Probability method), 36
get_monomials() (in module ncpol2sdpa), 35
get_neighbors() (in module ncpol2sdpa), 37
get_next_neighbors() (in module ncpol2sdpa), 37
get_relaxation() (ncpol2sdpa.FaacetsRelaxation method), 34
get_relaxation() (ncpol2sdpa.SdpRelaxation method), 25
get_relaxation() (ncpol2sdpa.SteeringHierarchy method), 28, 31
get_sos_decomposition() (ncpol2sdpa.SdpRelaxation method), 25

M

maximum_violation() (in module ncpol2sdpa), 38

P

pauli_constraints() (in module ncpol2sdpa), 36
Probability (class in ncpol2sdpa), 36
process_constraints() (ncpol2sdpa.SdpRelaxation method), 26
process_constraints() (ncpol2sdpa.SteeringHierarchy method), 29, 32
projective_measurement_constraints() (in module ncpol2sdpa), 37

R

read_sdpa_out() (in module ncpol2sdpa), 35

S

save_monomial_index() (ncpol2sdpa.SdpRelaxation method), 26
save_monomial_index() (ncpol2sdpa.SteeringHierarchy method), 29, 33
SdpRelaxation (class in ncpol2sdpa), 23
set_objective() (ncpol2sdpa.SdpRelaxation method), 26
set_objective() (ncpol2sdpa.SteeringHierarchy method), 29, 33
solve() (ncpol2sdpa.FaacetsRelaxation method), 34
solve() (ncpol2sdpa.SdpRelaxation method), 26
solve() (ncpol2sdpa.SteeringHierarchy method), 30, 33

SteeringHierarchy (class in ncpol2sdpa), 27, 30

W

write_to_file() (ncpol2sdpa.SdpRelaxation method), 27
write_to_file() (ncpol2sdpa.SteeringHierarchy method), 30, 33