
Ncpol2sdpa Documentation

Release 1.6

Peter Wittek

December 22, 2014

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Copyright and License | 1 |
| 1.2 | Acknowledgment | 1 |
| 2 | Download and Installation | 3 |
| 2.1 | Dependencies | 3 |
| 2.2 | Installation | 3 |
| 3 | Examples | 5 |
| 3.1 | Example 1: Toy Example | 5 |
| 3.2 | Example 2: Using MOSEK or PICOS | 7 |
| 3.3 | Example 3: Mixed-Level Relaxation of a Bell Inequality | 7 |
| 3.4 | Example 4: Bosonic System | 8 |
| 3.5 | Example 5: Using the Nieto-Silleras Hierarchy | 9 |
| 3.6 | Example 6: Using the Moroder Hierarchy | 10 |
| 3.7 | Example 7: Sparse Relaxation with Chordal Extension | 11 |
| 4 | Revision History | 13 |
| 5 | Function Reference | 15 |
| 5.1 | SdpRelaxation Class | 15 |
| 5.2 | Functions to Work with SDPA, PICOS, and MOSEK | 16 |
| 5.3 | Functions to Help Define Polynomial Optimization Problems | 17 |
| 5.4 | Functions to Define Physics Problems | 17 |
| 6 | References | 21 |
| | Index | 23 |

INTRODUCTION

Ncpol2sdpa is a tool to convert a polynomial optimization problem of either commutative or noncommutative variables to a sparse semidefinite programming (SDP) problem that can be processed by the [SDPA](#) family of solvers, [MOSEK](#), or further processed by [PICOS](#) to solve the problem by [CVXOPT](#). 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. Example applications include:

- [Ground-state energy problems](#): bosonic and fermionic systems, Pauli spin operators.
- [Maximum quantum violation of Bell inequalities](#), also in [multipartite scenarios](#).
- [Nieto-Silleras hierarchy](#) for quantifying randomness.
- [Moroder hierarchy](#) to enable PPT-style and other additional constraints.
- If using commutative variables, the hierarchy is identical to [Lasserre's](#).

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:

Wittek, P. (2014). [Ncpol2sdpa – Sparse Semidefinite Programming Relaxations for Polynomial Optimization Problems of Noncommuting Variables](#). To Appear in ACM Transactions on Mathematical Software.

1.1 Copyright and License

Copyright © 2012-2014 P. Wittek

Ncpol2sdpa is free software; you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

Ncpol2sdpa is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the [GNU General Public License](#) for more details.

1.2 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](#) project number SNIC 2014/2-7.

DOWNLOAD AND INSTALLATION

The entire package for is available as a [gzipped tar file](#) from the [Python Package Index](#), containing the source, documentation, and examples.

The latest development version is available on [GitHub](#).

2.1 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](#).

Optional dependencies include:

- [SciPy](#) allows faster execution with the default CPython interpreter, and enables removal of equations and chordal graph extensions.
- [Chompack](#) improves the sparsity of the chordal graph extension.
- [PICOS](#) is necessary for converting the problem to a PICOS problem.
- [MOSEK](#) Python module is necessary to work with the MOSEK converter.
- [Cvxopt](#) is required by both Chompack and PICOS.

2.2 Installation

Follow the standard procedure for installing Python modules:

```
$ pip install ncpol2sdpa --user
```

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 --user
```


EXAMPLES

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.

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

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

Getting the relaxation requires at least the level of relaxation:

```
sdpRelaxation.get_relaxation(level)
```

This will generate the moment matrix. Additional elements of the problem, such as the objective function, inequalities, equalities, and bounds on the variables.

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 is also supported for writing a problem and solving it. Using a converter to PICOS, it is also possible to solve the problem with a range of other solvers, including CVXOPT.

3.1 Example 1: Toy Example

Consider the following polynomial optimization problem (Pironio, Navascués, and Acín 2010):

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

such that

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

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

Entering the objective function and the inequality constraint is easy. The equality constraint is a simple projection. We either substitute two inequalities to replace the equality, or treat the equality as a monomial substitution. The second

option leads to a sparser SDP relaxation. The code samples below take this approach. In this case, the monomial basis is $\{1, x_1, x_2, x_1x_2, x_2x_1, x_2^2\}$. The corresponding relaxation is written as

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

such that

$$\begin{bmatrix} 1 & y_1 & y_2 & y_{12} & y_{21} & y_{22} \\ y_1 & y_1 & y_{12} & y_{12} & y_{121} & y_{122} \\ y_2 & y_{21} & y_{22} & y_{212} & y_{221} & y_{222} \\ y_{12} & y_{121} & y_{122} & y_{1212} & y_{1221} & y_{1222} \\ y_{21} & y_{211} & y_{212} & y_{2121} & y_{2122} & y_{2123} \\ y_{22} & y_{221} & y_{222} & y_{2212} & y_{2221} & y_{2222} \end{bmatrix} \succeq 0$$

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

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

```
from ncpol2sdpa import *

# 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]

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

# Obtain SDP relaxation
sdpRelaxation = SdpRelaxation(X)
sdpRelaxation.get_relaxation(level, objective=obj, inequalities=inequalities,
                             substitutions=monomial_substitution)
write_to_sdpa(sdpRelaxation, 'exemplenc.dat-s')
```

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

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

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

```
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 $-3/4$.

3.2 Example 2: Using MOSEK or PICOS

Apart from SDPA, MOSEK also enjoys full support. Using the preliminaries of the problem outlined in Section [example1], 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 installation is operational.

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

```
P = convert_to_picos(sdpRelaxation)
```

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

```
P.solve()
```

3.3 Example 3: 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.

```
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)
```

Then we need to generate the monomials we would like to add to the relaxation.

```
AB = [Ai*Bj for Ai in flatten(A) for Bj in flatten(B)]
```

We have to tell when we ask for the relaxation that these extra monomials should be considered:

```
sdpRelaxation = SdpRelaxation(flatten([A, B]))
sdpRelaxation.get_relaxation(level, objective=objective,
                             substitutions=monomial_substitutions,
                             extramonomials=AB)
```

3.4 Example 4: 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 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(level, objective=hamiltonian,
                             equalities=equalities,
                             substitutions=substitutions,
                             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).

3.5 Example 5: 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:

```
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(level, objective=objective,
                            equalities=equalities,
                            substitutions=monomial_substitutions)

print(solve_sdp(sdpRelaxation))
```

3.6 Example 6: 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].

```
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(level, objective=objective,
                            substitutions=monomial_substitutions)
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.

3.7 Example 7: Sparse Relaxation with Chordal Extension

This method replicates the behaviour of SparsePOP (Waki et. al, 2008). It is invoked by defining the hierarchy as "npa_chordal". The following is a simple example:

```
level = 2
X = generate_variables(3, commutative=True)

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]

sdpRelaxation = SdpRelaxation(X, hierarchy="npa_chordal")
sdpRelaxation.get_relaxation(level, objective=obj, inequalities=inequalities)
print(solve_sdp(sdpRelaxation))
```


REVISION HISTORY

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](https://doi.org/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 `nextraobjvars=[]` 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](https://doi.org/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](https://doi.org/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.

FUNCTION REFERENCE

5.1 SdpRelaxation Class

class `ncpol2sdpa.SdpRelaxation` (*variables*, *nonrelaxed=None*, *verbose=0*, *hierarchy='npa'*, *normalized=True*)
 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 nonhermitian variables, possibly a list of list of variables if the hierarchy is not NPA.
- **nonrelaxed** (list of `sympy.physics.quantum.operator.Operator` or `sympy.physics.quantum.operator.HermitianOperator` or a list of list.) – Optional variables which are not to be relaxed.
- **verbose** (*int.*) – Optional parameter for level of verbosity:
 - 0: quiet
 - 1: verbose
 - 2: debug level
- **hierarchy** (*str.*) – Optional parameter for defining the type of hierarchy (default: “npa”):
 - “npa”: the standard NPA hierarchy ([doi:10.1137/090760155](https://doi.org/10.1137/090760155)). When the variables are commutative, this formulation is identical to the Lasserre hierarchy.
 - “npa_chordal”: chordal graph extension to improve sparsity ([doi:10.1137/050623802](https://doi.org/10.1137/050623802))
 - “nieto-silleras”: [doi:10.1088/1367-2630/16/1/013035](https://doi.org/10.1088/1367-2630/16/1/013035)
 - “moroder”: [doi:10.1103/PhysRevLett.111.030501](https://doi.org/10.1103/PhysRevLett.111.030501)
- **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.

get_relaxation (*level*, *objective=None*, *inequalities=None*, *equalities=None*, *substitutions=None*, *bounds=None*, *removeequalities=False*, *extramonomials=None*, *nsextraobjvars=None*)

Get the SDP relaxation of a noncommutative polynomial optimization problem.

Parameters

- **level** (*int.*) – The level of the relaxation

- **obj** (`sympy.core.expr.Expr`) – Optional parameter to describe the objective function.
- **inequalities** (list of `sympy.core.expr.Expr`) – Optional parameter to list inequality constraints.
- **equalities** (list of `sympy.core.expr.Expr`) – Optional parameter to list equality constraints.
- **substitutions** (dict of `sympy.core.expr.Expr`) – Optional parameter containing monomials that can be replaced (e.g., idempotent variables).
- **bounds** (list of `sympy.core.expr.Expr`) – Optional parameter of bounds on variables which will not be relaxed by localizing matrices.
- **removeequalities** (*bool*.) – Optional parameter to attempt removing the equalities by solving the linear equations.
- **extramonomials** (list of `sympy.core.expr.Expr`) – Optional parameter of monomials to be included, on top of the requested level of relaxation.
- **nsextraobjvars** (*list of float*.) – Optional parameter of the coefficients of unnormalized top left elements of the moment matrices of the Nieto-Silleras hierarchy that should be included in the objective function.

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

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

Parameters

- **objective** (`sympy.core.expr.Expr`) – Describes the objective function.
- **nsextraobjvars** (*list of float*.) – Optional parameter of the coefficients of unnormalized top left elements of the moment matrices of the Nieto-Silleras hierarchy that should be included in the objective function.

5.2 Functions to Work with SDPA, PICOS, and MOSEK

`ncpol2sdpa.solve_sdp` (*sdpRelaxation*, *solverexecutable*='sdpa')

Helper function to write out the SDP problem to a temporary file, call the solver, and parse the output.

Parameters

- **sdpRelaxation** (`ncpol2sdpa.SdpRelaxation`.) – The SDP relaxation to be solved.
- **solverexecutable** (*str*.) – Optional parameter to specify the name of the executable if sdpa is not in the path or has a different name.

Returns tuple of float – the primal and dual solution of the SDP, respectively.

`ncpol2sdpa.write_to_sdpa` (*sdpRelaxation*, *filename*)

Write the SDP relaxation to SDPA format.

Parameters

- **sdpRelaxation** (`ncpol2sdpa.SdpRelaxation`.) – The SDP relaxation to write.
- **filename** (*str*.) – The name of the file. It must have the suffix ".dat-s"

`ncpol2sdpa.convert_to_mosek` (*sdpRelaxation*)

Convert an SDP relaxation to a MOSEK task.

Parameters `sdpRelaxation` (`ncpol2sdpa.SdpRelaxation`.) – The SDP relaxation to convert.

Returns `mosek.Task`.

`ncpol2sdpa.convert_to_picos` (`sdpRelaxation`)

Convert an SDP relaxation to a PICOS problem.

Parameters `sdpRelaxation` (`ncpol2sdpa.SdpRelaxation`.) – The SDP relaxation to convert.

Returns `picos.Problem`.

5.3 Functions to Help Define Polynomial Optimization Problems

`ncpol2sdpa.generate_variables` (`n_vars`, `hermitian=False`, `commutative=False`, `name='x'`)

Generates a number of commutative or noncommutative variables

Parameters `n_vars` (*int.*) – The number of variables.

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

`ncpol2sdpa.get_ncmonomials` (`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.ncdegree` (*polynomial*)

Returns the degree of a noncommutative polynomial.

Parameters `polynomial` (`sympy.core.expr.Expr`.) – Polynomial of noncommutative variables.

Returns `int` – the degree of the polynomial.

`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.

5.4 Functions to Define Physics Problems

`ncpol2sdpa.bosonic_constraints` (*a*)

Return a set of constraints that define bosonic ladder operators.

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

Returns tuple of dict of substitutions and list of equalities.

`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 tuple of dict of substitutions and list of equalities and inequalities.

`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 neighbors of an operator 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.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)`

Get the maximum violation of a 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, A, B)`

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.
- **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 `sympy.core.expr.Expr` – the objective function to be solved by SDPA as minimization problem to find the maximum quantum violation.

REFERENCES

- Bancal, Jean-Daniel, Lana Sheridan, and Valerio Scarani. 2014. “More Randomness from the Same Data.” *New Journal of Physics* 16 (3): 033011. doi:[10.1088/1367-2630/16/3/033011](https://doi.org/10.1088/1367-2630/16/3/033011).
- Fayngold, M., and V. Fayngold. 2013. *Quantum Mechanics and Quantum Information*. Wiley-VCH.
- 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](https://doi.org/10.1016/j.cpc.2012.11.019).
- 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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1145/1377612.1377619).
- Yamashita, M., K. Fujisawa, and M. Kojima. 2003. “SDPARA: Semidefinite Programming Algorithm Parallel Version.” *Parallel Computing* 29 (8): 1053–67.

B

bosonic_constraints() (in module ncpol2sdpa), 17

C

convert_to_mosek() (in module ncpol2sdpa), 16

convert_to_picos() (in module ncpol2sdpa), 17

correlator() (in module ncpol2sdpa), 18

D

define_objective_with_I() (in module ncpol2sdpa), 19

F

fermionic_constraints() (in module ncpol2sdpa), 17

flatten() (in module ncpol2sdpa), 17

G

generate_measurements() (in module ncpol2sdpa), 18

generate_variables() (in module ncpol2sdpa), 17

get_ncmonomials() (in module ncpol2sdpa), 17

get_neighbors() (in module ncpol2sdpa), 18

get_relaxation() (ncpol2sdpa.SdpRelaxation method), 15

M

maximum_violation() (in module ncpol2sdpa), 19

N

ncdegree() (in module ncpol2sdpa), 17

P

pauli_constraints() (in module ncpol2sdpa), 18

projective_measurement_constraints() (in module ncpol2sdpa), 18

S

SdpRelaxation (class in ncpol2sdpa), 15

set_objective() (ncpol2sdpa.SdpRelaxation method), 16

solve_sdp() (in module ncpol2sdpa), 16

W

write_to_sdpa() (in module ncpol2sdpa), 16