Manual of *Quantum_Inequality* MATLAB package

By

Nguyen Truong Duy

Introduction

Quantum_Inequality is a MATLAB package to compute optimal upperbounds of Bell inequalities, and to extract optimal state and measurements if they exist. This manual serves as supplementary material to the package. It has two parts. The first part contains instructions for installing the package. The second part provides a detailed tutorial of how to use features in the package.

Table of Contents

Chapter 1

Installing

1.1 Prerequisites

The *Quantum_Inequality* package relies on the following third-party software which should be

- Matlab. The recommended version is $R2010a$ or above.
- Yalmip. The recommended version is $R20120109$ or above. The package can be downloaded at `http://users.isy.liu.se/johanl/yalmip/`
- A semidefinite programming (SDP) solver that Yalmip supports, such as SeDuMi, SDPT3, etc. A full list of SDP solvers supported by Yalmip can be found at `http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Solvers.Solvers`

  You can download Yalmip $R20120109$ and SeDeMi 1.3 at our code repository `https://github.com/truongduy134/Quantum_Inequality/tree/master/Third_party_software` and installed them as instructed.

1.2 Downloading and Installing

The package *Quantum_Inequality* is freely distributed at `https://github.com/truongduy134/Quantum_Inequality/tree/master/src`. After having all necessary third-party installed and added to your working directory, extracting the *Quantum_Inequality* package and copying it to your workspace, you should add these directories into your MATLAB path:

- /src
- /src/GenerateMonomial
- /src/NonCommuteMonomial
- /src/NonCommutePolynomial
- /src/ParsePolyExpr
- /src/RankLoopOptimalMeasure
- /src/TestCase (*optional*)
- /src/TestRoutine (*optional*)

You can add all these directories using the following single MATLAB command:

```
addpath(genpath(your_root_directory))
```

where *your_root_directory* is a string indicating the name of your root directory.

Chapter 2

Tutorial

In this chapter, we provide a brief tutorial of how to use features in the package.

2.1 Overview of general procedure and function calls

2.1.1 General procedure

Basically, there are 5 steps

1. Declare SDP solver settings (*optional*).
2. Describe the measurements, i.e. their names, which partition they belong to, which measurement setting they belong to (if they are projectors), etc. Declare the polynomial you want to optimize. Then call a function to create a polynomial object.
3. Declare the criteria for filtering monomials (*optional*)
4. Call the main routine to compute the optimal upperbound.
5. Call the routine to check if rank loop has occurred (for the projector case), and extract optimal state and measurements (if they exist).

2.1.2 Specifying SDP solver settings

To specify the solver settings, we use the function *sdpsettings* in Yalmip package with the following syntax:

```
options = sdpsettings('field_1', value_1, 'field_2', value_2, ... )
```

A detailed discussion of the function *sdpsettings* and a list of fields you can specify can be found in `http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Commands.sdpsettings`
This step is optional. If you do not specify solver settings, then the package uses the default setting, which is:

```
solverSpecify = sdpsettings('solver','sedumi','sedumi.eps',1e-12);
```

2.1.3 Creating a polynomial object

In order to find a quantum bound of the Bell inequality, we must provide information such as the polynomial we want to optimize, the variable type (observables or projectors), measurement settings and partitions. These pieces of information are encapsulated in the polynomial object.

To get a polynomial object representing the polynomial you want to optimize, you need to perform the following 3 steps:

1. Declare a variable to specify information regarding the variable names, partitions, and measurement settings. This variable is a cell C of size $1 \times n$. Each element of C is again a cell C_i representing a partition (or party) of size $1 \times n_{C_i}$. There are two cases:

   - If your measurements are observables, each element of C_i is a string indicating the name of a variable belonging to that partition.
   - If your measurements are projectors, each element of C_i is a cell C_{ij} of size $1 \times m_{C_{ij}}$ representing a measurement setting in that partition. Each element of C_{ij} is a string indicating the name of a variable belonging to that measurement setting.

   A variable name should follow MATLAB variable naming conventions. The name can also have other symbols, except $+$, $-$, ^, $/$, $*$. The name should not contain white spaces.

Example 2.1.1. *Consider the Yao Inequality where* $\{A_1, A_2, A_3\}$, $\{B_1, B_2, B_3\}$, $\{C_1, C_2, C_3\}$ *are observables belonging to 3 partitions respectively. We can specify a MATLAB variable indicating the pieces information regarding measurements of Yao Inequality as follows:*

```
varPropWithName = {{'A_1', 'A_2', 'A_3'}, {'B_1', 'B_2', 'B_3'}, {'C_1', ...
    'C_2', 'C_3'}};
```

Example 2.1.2. *Consider the Mod-3 Game where there are two parties Alice and Bob receiving inputs* s, t *from the sets* $S = T = \{0, 1, 2\}$ *respectively, and producing outputs* a, b *in the sets* $A = B = \{0, 1, 2\}$. *Alice and Bob win the game if* $a + b \equiv st \bmod 3$. *Let* $\{A_s^a\}$ *and* $\{B_t^b\}$ *be sets of projectors of Alice and Bob respectively. Then we can specify a MATLAB variable indicating the pieces information regarding measurements of the Mod-3 Game as follows:*

```
varPropWithName = {{{'As0a0', 'As0a1', 'As0a2'}, {'As1a0', 'As1a1', ...
    'As1a2'}, {'As2a0', 'As2a1', 'As2a2'}}, {{'Bt0b0', 'Bt0b1', 'Bt0b2'}, ...
    {'Bt1b0', 'Bt1b1', 'Bt1b2'}, {'Bt2b0', 'Bt2b1', 'Bt2b2'}}};
```

2. Next, you provide a MATLAB string representing the infix expression of the polynomial you want to optimize. The package recognizes the following operations in the infix expression: $+$ (addition), $-$ (subtraction), $*$ (multiplication), $/$ (division), ^ (exponentiation). Only parentheses () are recognized in the package. Operator precedence follows from mathematical conventions. The package does not support division between polynomials of degree at least 1. The name of variables in the expression should be consistent with those you specified in the previous step.

Example 2.1.3. *Let us continue with Example 2.1.1. The Yao Inequality can be written as* $B_{Yao} = A_1B_2C_3 + A_2B_3C_1 + A_3B_1C_2 - A_1B_3C_2 - A_2B_1C_3 - A_3B_2C_1$. *You can specify its infix expression as follows:*

```
polyStr = 'A_1 * B_2 * C_3 + A_2 * B_3 * C_1 + A_3 * B_1 * C_2 - A_1 * B_3 * ...
    C_2 - A_2 * B_1 * C_3 - A_3 * B_2 * C_1';
```

Example 2.1.4. *Let us continue with Example 2.1.2. The infix expression of the polynomial of the Mod-3 Game* $B_{Mod-3} = \frac{1}{9}(A_0^0(B_0^0 + B_1^0 + B_2^0) + A_0^1(B_0^2 + B_1^2 + B_2^2) + A_0^2(B_0^1 + B_1^1 + B_2^1) + A_1^0(B_0^0 + B_1^1 + B_2^2) + A_1^1(B_0^2 + B_1^0 + B_2^1) + A_1^2(B_0^1 + B_1^2 + B_2^0) + A_2^0(B_0^0 + B_1^2 + B_2^1) + A_2^1(B_0^2 + B_1^1 + B_2^0) + A_2^2(B_0^1 + B_1^0 + B_2^2))$ *can be specified as:*

```
polyStr = '1/9 * (As0a0 * (Bt0b0 + Bt1b0 + Bt2b0) + As0a1 * (Bt0b2 + Bt1b2 + ...
    Bt2b2) + As0a2 * (Bt0b1 + Bt1b1 + Bt2b1) + As1a0 * (Bt0b0 + Bt1b1 + ...
    Bt2b2) + As1a1 * (Bt0b2 + Bt1b0 + Bt2b1) + As1a2 * (Bt0b1 + Bt1b2 + ...
    Bt2b0) + As2a0 * (Bt0b0 + Bt1b2 + Bt2b1) + As2a1 * (Bt0b2 + Bt1b1 + ...
    Bt2b0) + As2a2 * (Bt0b1 + Bt1b0 + Bt2b2))';
```

3. Finally, you provoke the function *createPolyFromExpr* with the parameters you have created in the two previous steps to get the polynomial object representing your polynomial The prototype of *createPolyFromExpr* is as follows:

```
function [polyObj, reducedVarHash] = createPolyFromExpr(expr, ...
    varPropWithName, varTypeName, isFull)
```

The following is to explain the function parameters.

- *expr* is the infix expression of the polynomial.
- $varPropWithName$ is the variable specifying properties of the variables in the polynomial that you declare in the first step.
- $varTypeName$ can be either ***'observable'*** or ***'projector'***, which is to specify the type of measurements in the polynomial.
- The final parameter $isFull$ is optinal because it is only needed in the case where measurements are projectors. If variables are observables, it will be ignored. The parameter can be either ***'full'*** or ***'partial'***. If you indicate the final parameter to be $'full'$, it means that projectors in each measurement setting can establish the identity equation, i.e. their sum equals to the identity matrix (Note that our measurements are matrices).

The function returns two variables.

- *polyObj* is the polynomial object representing your polynomial.
- $reducedVarHash$ is a hash table that keeps track of redundant projectors in case your measurement settings are full.

Example 2.1.5. *To get a polynomial object of Yao Inequality in Example 2.1.1, we call*

```
[polyObj ~] = createPolyFromExpr(polyStr, varPropWithName, 'observable');
```

Similarly, to get a polynomial object of Mod-3 Inequality in Example 2.1.2, we call

```
[polyObj reduceVar] = createPolyFromExpr(polyStr, varPropWithName, ...
    'projector', 'full');
```

2.1.4 Declaring criteria for filtering monomials

The $Quantum_Inequality$ package allows you to specify the criteria for generating the list of monomials which is used to index the moment matrix. This can be done by calling the function $specifyGenListMonoCriteria$. Its prototype is:

```
function hashGenMonoInfo = specifyGenListMonoCriteria(your_criteria)
```

where *your_criteria* is a cell of size $1 \times k$. Each element of $your_criteria$ is again a cell E such that

- Its first element $E\{1\}$ is an integer indicating the length (or degree) of a monomial.
- Its second element $E\{2\}$ is an integer indicating the minimum number of partition that should be present in a monomial (i.e. a monomial generated of degree $E\{1\}$ should contain at least $E\{2\}$ instances of variables from $E\{2\}$ different partitions). $E\{2\} = 0$ means no restriction on monomials in terms of the number of partitions present.
- Its third element $E\{3\}$ is a string whose value is either ***'full'*** or ***'random'***. If $E\{3\}$ is ***'full'***, we take all monomials that satisfy $E\{1\}$ and $E\{2\}$. If $E\{3\}$ is ***'random'***, we choose randomly some elements from the list of all monomials satisfying $E\{1\}$ and $E\{2\}$.
- $E\{4\}$ and $E\{5\}$ are needed if $E\{3\}$ is ***'random'*** to specify how a subset of monomials is randomly chosen. $E\{5\}$ is either ***'absolute'*** or ***'ratio'***. If $E\{5\}$ is ***'absolute'***, $E\{4\}$ is an integer indicating the maximum number of monomials randomly chosen. If $E\{5\}$ is either ***'ratio'***, $E\{4\}$ is a rational number indicating the percentage of monomials randomly chosen over the total number of monomials.

and $hashGenMonoInfo$ is a hash table that keeps track of your monomial-generating criteria returned by the function. This will be passed as a parameter to the main routine when finding an optimal upperbound of a polynomial.

For monomials of degree d that you do not specify generating criteria in *your_criteria*, there is no restriction on them and the program takes all of them. The step of specifying monomial-generating criteria is optional. If no criterion is specified, all monomials of degree up to the level of semidefinite program you are running at are chosen.

Example 2.1.6. *Suppose you will run the quantum bound rountine at level 3. For the list of monomials used to index the moment matrix, you want to take all monomials of degree up to 2. Because the number of monomials of degree 3 may be large, you want to take at most 30 monomials of degree 3 in which there are variables from at least 2 partitions present. Then you can specify these pieces of information as follows:*

```
hashGenMonoInfo = specifyGenListMonoCriteria({{3, 2, 'random', 30, ...
    'absolute'}});
```

Example 2.1.7. *Now suppose you run the quantum bound rountine at level 5. For the list of monomials used to index the moment matrix, you want to take all monomials of degree up to 2. You want to take all monomials of degree 3 in which there are variables from at least 2 partitions present. For monomials of level 4 and 5, you want to take randomly $\frac{1}{9}$ and $\frac{1}{10}$ of the total number of monomials of level 4 and 5 respectively (there is no restriction on the number of partitions present). Then you can specify these pieces of information as follows:*

```
hashGenMonoInfo = specifyGenListMonoCriteria({{3, 2, 'full'}, {5, 0, ...
    'random', 1/10, 'ratio'}, {4, 0, 'random', 1/9, 'ratio'}});
```

2.1.5 Finding quantum bounds

The main routine of the package is *findQuantumBound*, which is to find a quantum bound of a Bell inequality based on the SDP (Semidefinite Programmign) relaxation algorithm. Its protoype is:

```
function [upperBoundVal solverMessage momentMatrix monoMapTable monoList] = ...
    findQuantumBound(polyObj, sdpLvl, hashGenMonoInfo, solverSetting);
```

The input parameters of *findQuantumBound* are:

- *polyObj*, *hashGenMonoInfo*, *solverSetting* are variables you have declared in the above steps. Note that *hashGenMonoInfo* and *solverSetting* are optional and can be ignored.
- *sdpLvl* is an integer indicating the level of the SDP you run at. Note that $2 \times sdpLvl \geq \deg(polyObj)$ where $\deg(polyObj)$ is the degree of the polynomial.

The function returns a list of variables

- *upperBoundVal* is a number indicating an upperbound of the input Bell inequality at the level *sdpLvl*.
- *solverMessage* is a structure keeping track of the SDP solver's message.
- *momentMatrix*, *monoMapTable*, *monoList* are some variables which will be passed to other functions when detecing rank loop, or when extracting optimal state and measurements.

2.1.6 Extracting optimal states and measurement

In case where the Bell inequality involves only 2 parties (or partitions), the package has routines to detect if the upper bound at a particular SDP level is optimal, and to extract optimal state and measurements. In this subsection, we go through how to use those features of the package.

Observable case

In the observable case, the upperbound at level 1 converges to the optimal quantum bound. Therefore, there is no need to detect if the convergence to the optimal value occurs. We can extract the optimal state and observable measurements using the function *getOptimalObservable*. Its prototype is

```
function [cellObservable opState] = getOptimalObservable(momentMatrix, ...
    polyObj, monoMapTable)
```

The inputs of the function are *momentMatrix* and *monoMapTable* which are returned by the main routine *findQuantumBound*, and *polyObj* which is the polynomial object representing your polynomial. The function *getOptimalObservable* returns *cellObservable* which is a cell of matrix representations of the optimal observables, and *opState* which is the optimal state.

Projector case

In the projector case, we must first check to see if rank loop occurs. This can be done with the following function

```
function [rankLoop rankMoment epsilon] = hasRankLoop(momentMatrix, sdpLvl, ...
    monoList, polyObj)
```

The inputs of the function are $momentMatrix$, $monoList$ which are returned by the main routine $findQuantumBound$, and *polyObj* is a polynomial object, and *sdpLvl* is the SDP level you run in the main routine. The function returns a list of variables:

- $rankLoop = 1$ if rank loop occurs, i.e. the upperbound at the level *sdpLvl* is indeed an optimal upperbound of the Bell inequality. Otherwise $rankLoop = 0$.
- $rankMoment$ is the rank of the moment matrix.
- *epsilon* is a zero threshold (i.e. for any real number x, if $|x| < epsilon$, the package treats x as 0) that rank loop occurs. If $rankLoop = 1$, $epsilon \leq 10^{-6}$.

If rank loop occurs, we can extract the optimal state and projector measurements using the function $getOptimalProjector$. Its prototype is:

```
function [cellProjector opState] = getOptimalProjector(momentMatrix, sdpLvl, ...
    listMono, polyObj)
```

The inputs of the function are the same as the ones in $hasRankLoop$. The function $getOptimalProjector$ returns $cellProjector$ which is a cell of matrix representations of the optimal projectors, and *opState* which is the optimal state.

```
function [cellProjector opState] = getOptimalProjector(momentMatrix, sdpLvl, ...
    listMono, polyObj)
```

Formatting the measurements

For you to easily work with the matrix value of your measurements, the package provide a routine to map a measurement name to its corresponding matrix value returned by *getOptimalObservable* and $getOptimalProjector$. This can be done by calling the function $formatOutputMeasure$. Its prototype is:

```
function hashVarNameVal = formatOutputMeasure(cellVarValue, varPropWithName, ...
    varType)
```

where

- $cellVarValue$ is a cell of matrix values returned by *getOptimalObservable* or $getOptimalProjector$.
- $varPropWithName$ is a variable specifying information regarding the measurements (their names, partitions, measurement settingsm, etc.) that you have declared in Subsection 2.1.3.
- $varType$ is either ***'observable'*** or ***'projector'***.

- *hashVarNameVal* is a hash table that maps a string indicating the measurement name to its matrix value.

2.2 Tutorial with Observable Measurements

In this section, we provide a detailed tutorial of how to use the package to compute an optimal upperbound of a Bell inequality whose measurements are observables. We also illustrate how to extract the optimal state and measurements.

Let us consider the problem of finding an optimal upperbound of the classic $CHSH$ inequality. The polynomial we want to optimize is

$$B_{CHSH} = A_1 B_1 + A_1 B_2 + A_2 B_1 - A_2 B_2 \tag{2.1}$$

where A_1, A_2 are observables of the first party, and B_1, B_2 are observables of the second party.

2.2.1 Sample Code

In this sample code of computing the optimal upperbound of $CHSH$ inequality, we use the default solver settings, and use all monomials of length 1.

Step 1: Omitted

Step 2: We declare the observable measurements, the polynomial expression of $B_{CHSH} = A_1 B_1 + A_1 B_2 + A_2 B_1 - A_2 B_2$ (variables are observables) and provoke the function to create the polynomial object.

```
varPropWithName = {{'A_1', 'A_2'}, {'B_1', 'B_2'}};
polyStr = 'A_1 * B_1 + A_1 * B_2 + A_2 * B_1 - A_2 * B_2';
[polyObj ¬] = createPolyFromExpr(polyStr, varPropWithName, 'observable');
```

Step 3: Omitted

Step 4: Provoke the main routine to compute the optimal upperbound.

```
sdpLvl = 1;
[upperBoundVal solverMessage momentMatrix monoMapTable monoList] = ...
    findQuantumBound(polyObj, sdpLvl);
disp('Upperbound value = ');
disp(upperBoundVal);
```

Step 5: Extract the optimal state and observables. Then map the observable name to its matrix value.

```
[cellObservable opState] = getOptimalObservable(momentMatrix, polyObj, ...
    monoMapTable);
hashVarNameVal = formatOutputMeasure(cellObservable, varPropWithName, ...
    'observable')
```

Then from now on, when we want to examine the matrix value of a certain measurement, for instance 'A_1', we just need to type

```
hashVarNameVal('A_1')
```

2.3 Tutorial with Projector Measurements

In this section, we provide a detailed tutorial of how to use the package to compute an optimal upperbound of a Bell inequality whose measurements are projectors. We also illustrate how to detect if rank loop occursa and to extract the optimal state and measurements.

2.3.1 Sample code of $I3322$

Step 1: Omitted

Step 2: We declare the observable measurements, the polynomial expression of $B_{I3322=A_1^a(B_1^b+B_2^b+B_3^b)+A_2^a(B_1^b+B_2^b}$ and provoke the function to create the polynomial object.

```
varPropWithName = {{{'A"a_1'}, {'A"a_2'}, {'A"a_3'}}, {{'B"b_1'}, ...
    {'B"b_2'}, {'B"b_3'}}};
expr = 'A"a_1 * (B"b_1 + B"b_2 + B"b_3) + A"a_2 * (B"b_1 + B"b_2 - ...
    B"b_3) + A"a_3 * (B"b_1 - B"b_2) - A"a_1 - 2 * B"b_1 - B"b_2';
polyObj = createPolyFromExpr(expr, varPropWithName, 'projector', ...
    'partial');
```

Step 3: We will run the main routine at level 3. We want to take all monomials of degree 3 in which there are variables from at least 2 partitions present

```
hashGenMonoInfo = specifyGenListMonoCriteria({{3, 2, 'full'}});
```

Step 4: Provoke the main routine to compute the optimal upperbound.

```
sdpLvl = 3;
[upperBoundVal solverMessage momentMatrix monoMapTable monoList] = ...
    findQuantumBound(polyObj, sdpLvl, hashGenMonoInfo);
disp('Upperbound value = ');
disp(upperBoundVal);
```

Step 5: Check if the rank loop has occured.

```
[rankLoop rankMoment epsilon] = hasRankLoop(momentMatrix, sdpLvl, ...
    monoList, polyObj);
if rankLoop
    disp('Rank loop occurs with threshold = ');
    disp(epsilon);
else
    disp('Rank loop does NOT occur');
end
```

2.3.2 Sample code of Mod-2 Game

Step 1: Omitted

Step 2: We declare the observable measurements, the polynomial expression of $B_{Mod-2} = \frac{1}{4}(A_0^1 B_0^1 + A_0^0 B_0^0 + A_0^0 B_1^0 + A_0^1 B_1^1 + A_1^0 B_0^0 + A_1^0 B_1^1 + A_1^1 B_1^0 + A_1^1 B_0^1)$ where $\{A_0^1, A_0^0\}$, $\{A_1^0, A_1^1\}$ and provoke the function to create the polynomial object.

```
polyStr = '1/4 * (A1_0 * B1_0 + A0_0 * B0_0 + A0_0 * B0_1 + A1_0 * B1_1 ...
    + A0_1 * B0_0 + A0_1 * B1_1 + A1_1 * B0_1 + A1_1 * B1_0)';
varPropWithName = {{{'A0_0', 'A1_0'}, {'A0_1', 'A1_1'}}, {{'B0_0', ...
    'B1_0'}, {'B0_1', 'B1_1'}}};
[polyObj reduceVar] = createPolyFromExpr(polyStr, varPropWithName, ...
    'projector', 'full');
```

Step 3: Omitted. In this sample code, we extract optimal state and projectors of B_{Mod-2}. Therefore, we need to run the main rountine at the full level (Rank loop and optimizer extraction are not available to intermediate levels).

Step 4: Provoke the main routine at level 3 to compute the optimal upperbound.

```
sdpLvl = 3;
[upperBoundVal solverMessage momentMatrix monoMapTable monoList]  = ...
    findQuantumBound(polyOp, sdpLvl);
disp('Upperbound value = ');
disp(upperBoundVal);
```

Step 5: Check if rank loop occurs. If so, extract the optimal state and projectors

```
[rankLoop rankMoment epsilon] = hasRankLoop(momentMatrix, sdpLvl, ...
    monoList, polyObj);
% Print rank loop result
if rankLoop
    disp('Rank loop occurs with threshold = ');
    disp(epsilon);
else
    disp('Rank loop does NOT occur');
end
% Extract optimal state and projectors
if rankLoop
    [cellProjector opState] = getOptimalProjector(momentMatrix, sdpLvl, ...
        listMono, polyObj);
    %Map variable names to their matrix values
    hashVarNameVal = formatOutputMeasure(cellProjector, ...
        varPropWithName, 'projector');
end
% Do something fun with your optimal projectors (e.g. examining their ...
    properties).
t = trace(hashVarNameVal('A1_0')' * hashVarNameVal('B1_0'))
```