"Device-independent lower bounds on the conditional von Neumann entropy" – User guide

Peter Brown*

September 21, 2021

Let's begin by recapping what we mean by a rate of a device-independent protocol. Alice and Bob have black box devices with inputs X and Y and outputs A and B respectively. We assume that the devices operate by measuring their parts of some tripartite state $\rho_{Q_A Q_B Q_E}$ where Q_E is the system of some potential adversary. In particular, we assume the conditional distribution can be written as

$$p(a,b|x,y) = \mathrm{Tr}\left[\rho_{Q_A Q_B Q_E}(M_{a|x} \otimes N_{b|y} \otimes I)\right].$$

A possible configuration of the devices is described by a tuple $(Q_A Q_B Q_E, \rho, \{M_{a|x}\}, \{N_{b|y}\})$ which we refer to as a *strategy*. To each strategy we can associate a post-measurement state

$$\rho_{ABXYQ_E} = \sum_{abxy} p(xy)|abxy\rangle\langle abxy| \otimes \rho_{Q_E}(a,b,x,y)$$

where

$$\rho_{Q_E}(a,b,x,y) = \mathrm{Tr}_{Q_A Q_B}\left[\rho_{Q_A Q_B Q_E}(M_{a|x} \otimes N_{b|y} \otimes I)\right].$$

After fixing some linear constraints $\mathcal{R}$ on the conditional distribution of the devices, e.g., a Bell-inequality violation, we want to know the minimum conditional entropy produced by any devices satisfying those constraints. In particular the *rate* of a DI protocol is given by an expression of the form

$$r(\mathcal{R}) = \inf_{\text{strategies}|_{\mathcal{R}}} H(A|X=x, Q_E) \tag{1}$$

where the conditional von Neumann entropy is evaluated on the post-measurement state of a strategy and the infimum is over all strategies that are compatible with the constraints $\mathcal{R}$. Note that depending on the protocol we may optimize over $H(A|Q_E), H(A|XQ_E), H(AB|X=x, y=y, Q_E)$ or others (the numerical method can handle this).

We'll now detail how we can use our method to compute lower bounds on these rate problems. However, we'll illustrate most of the technique using the simpler example of the min-entropy. Almost everything is the same and in particular, if you can apply the min-entropy techniques then it is very likely you can apply our more complex method to get (hopefully) better results.

1 The min entropy

The min-entropy of the post-measurement state of a strategy above can be written as

$$H_{\min}(A|X=x, Q_E) = -\log G(A|X=x, Q_E)$$

*If you have further questions you can email me at peter.brown@ens-lyon.fr

where

$$\begin{aligned} G(A|X=x,Q_E) = \sup_{Z_a} \quad & \sum_a \mathrm{Tr}\left[\rho_{Q_A Q_B Q_E}(M_{a|x}\otimes I\otimes Z_a)\right] \\ \text{s.t.} \quad & \sum_a Z_a = I \\ & Z_a \geq 0 \qquad \text{for all } a. \end{aligned}$$

The constraints of this optimization problem encode that Z_a is a POVM on Eve's system. The overall quantity G can be interpreted as the optimal probability with which Eve can guess the outcome of Alice's measurement by measuring her part of the state.

$H_{\min}$ is never larger than H and so we can replace H with $H_{\min}$ in (1) in order to find lower bounds on the rates. Suppose the constraints in $\mathcal{R}$ all take the form $\sum_{abxy} c_{abxyj} p(a,b|x,y) \geq w_j$ for some $c_{abxyj}, w_j \in \mathbb{R}$. After doing so, we are left to solve the problem

$$\begin{aligned} \sup_{Z_a} \quad & \sum_a \mathrm{Tr}\left[\rho_{Q_A Q_B Q_E}(M_{a|x}\otimes I\otimes Z_a)\right] \\ \text{s.t.} \quad & \sum_a Z_a = I \\ & Z_a \geq 0 \\ & \sum_a M_{a|x} = \sum_b N_{b|y} = I \qquad \text{for all } x,y \\ & M_{a|x} \geq 0, \quad N_{b|y} \geq 0 \qquad \text{for all } a,b,x,y \\ & \sum_{abxy} c_{abxyj} \mathrm{Tr}\left[\rho_{Q_A Q_B Q_E}(M_{a|x}\otimes N_{b|y}\otimes I)\right] \geq w_j \qquad \text{for all } j. \end{aligned} \tag{2}$$

1.1 Relaxing to a noncommutative polynomial optimization problem

There's no known efficient method to compute the above problem. Thus we relax it to another type of problem. Namely the following

$$\begin{aligned} \sup_{Z_a} \quad & \sum_a \mathrm{Tr}\left[\rho M_{a|x} Z_a\right] \\ \text{s.t.} \quad & \sum_a Z_a = I \\ & Z_a \geq 0 \\ & \sum_a M_{a|x} = \sum_b N_{b|y} = I \qquad \text{for all } x,y \\ & M_{a|x} \geq 0, \quad N_{b|y} \geq 0 \qquad \text{for all } a,b,x,y \\ & \sum_{abxy} c_{abxyj} \mathrm{Tr}\left[\rho M_{a|x} N_{b|y}\right] \geq w_j \qquad \text{for all } j \\ & [M_{a|x}, N_{b|y}] = [M_{a|x}, Z_c] = [N_{b|y}, Z_c] = 0 \qquad \text{for all } a,b,x,y,c. \end{aligned} \tag{3}$$

Note that any feasible point of (2) defines a feasible point of (3) with the same objective value and so (3) we say *weaker* than (2). Importantly, the objective value of (3) is at least as large as (2) as we are optimizing over a larger set. In turn we get a lower bound on $H_{\min}$ (and the rates).To go from (2) to (3) we just remove the tensor product structure of our joint Hilbert space and impose that any operators that acted only on subspaces of the original space must now commute.

The above problem is what we refer to as a noncommutative polynomial optimization problem (NCPOP). We refer to any constraints that involve the trace as *moment constraints* and those that do not involve the trace as *operator constraints*.

The main reason we care about NCPOPs is that we have a technique (the NPA hierarchy [1]) that allows us to relax these optimizations further to semidefinite programs. These semidefinite programs we can then compute to get lower bounds on the rates of protocols!

2 Our new method

Our method is similar in many aspects to the min-entropy. We define a sequence of optimization problems that are lower bounds on the von Neumann entropy. Then, like the min-entropy approach, one can define a lower bound on the rate by replacing H with one of these optimizations. We then relax the optimization to NCPOP and then further to an SDP using the NPA hierarchy.

Here is the statement of our result.

Theorem 1. *Let $m \in \mathbb{N}$ and let t_i, w_i be the nodes and weights of an m-point Gauss-Radau quadrature rule with $t_m = 1$. Then (in the DI setting above) $H(A|X = x, Q_E)$ is bounded below by*

$$c_m + \sum_{i=1}^{m-1} \frac{w_i}{t_i \ln 2} \sum_a \inf_{Z_a \in B(Q_E)} \mathrm{Tr}\left[\rho_{Q_A Q_E}\left(M_{a|x^*} \otimes (Z_a + Z_a^* + (1-t_i) Z_a^* Z_a) + t_i (I_{Q_A} \otimes Z_a Z_a^*)\right)\right] \quad (4)$$
$$\text{s.t.} \quad \|Z_a\| \leq \alpha_i$$

where $c_m = \sum_{i=1}^{m-1} \frac{w_i}{t_i \ln 2}$.

Remark 1.

- The $H(A|X = x, Q_E)$ in the statement is defined for a particular strategy. In particular $\rho_{Q_A Q_B Q_E}$ would be the state of that strategy and $\{M_{a|x}\}_a$ would be Alice's POVM on input x.
- The statement is quite similar to the min-entropy statement in its form. However, the Z_a operators for Eve's system are no longer POVM elements. They are only bounded linear operators (not even Hermitian), they do have a norm constraint though!
- The w_i and t_i are somewhat common objects in numerical integration. In particular there are several packages to compute these (see the example scripts for one example).

Importantly we can also apply our method to conditional entropies other than $H(A|X = x, Q_E)$. To get a lower bound on $H(AB|X = x^*, Y = y^*, Q_E)$ you can replace the inner summation in (4) with

$$\sum_{ab} \inf_{Z_{ab} \in B(Q_E)} \mathrm{Tr}\left[\rho_{Q_A Q_B Q_E}\left(M_{a|x^*} \otimes N_{b|y^*} \otimes (Z_{ab} + Z_{ab}^* + (1-t_i) Z_{ab}^* Z_{ab}) + t_i (I_{Q_A Q_B} \otimes Z_{ab} Z_{ab}^*)\right)\right]. \quad (5)$$

(This form should be expected to those who know what the corresponding min-entropy form would be.) Similarly, to get a lower bound on $H(A|XQ_E)$ you can replace the inner sum with

$$\sum_{ax} \inf_{Z_{ax} \in B(Q_E)} p(x) \mathrm{Tr}\left[\rho_{Q_A Q_E}\left(M_{a|x} \otimes (Z_{ax} + Z_{ax}^* + (1-t_i) Z_{ax}^* Z_{ax}) + t_i (I_{Q_A} \otimes Z_{ax} Z_{ax}^*)\right)\right]. \quad (6)$$

Further expressions can be derived accordingly (just follow the proof of the corresponding lemma in the paper).

Because the form of (4) is similar to (1) we can do the same procedure: (1) replace H with our new lower bound in the rate optimization; (2) relax to a noncommutative polynomial optimization problem (remove tensor product structure and include commutation constraints); (3) relax the final problem to an SDP using

the NPA hierarchy and solve. For reference, the NCPOP for (4) is

$$\begin{aligned}
c_m + \inf \quad & \sum_{i=1}^{m-1} \frac{w_i}{t_i \ln 2} \sum_a \langle\psi| M_{a|x^*} (Z_{a,i} + Z_{a,i}^* + (1-t_i) Z_{a,i}^* Z_{a,i}) + t_i Z_{a,i} Z_{a,i}^* |\psi\rangle \\
\text{s.t.} \quad & \sum_{abxy} c_{abxyj} \langle\psi| M_{a|x} N_{b|y} |\psi\rangle \geq w_j && \text{for all } 1 \leq j \leq r \\
& \sum_a M_{a|x} = \sum_b N_{b|y} = I && \text{for all } x, y \\
& M_{a|x} \geq 0, \quad N_{b|y} \geq 0 && \text{for all } a, b, x, y \\
& Z_{a,i}^* Z_{a,i} \leq \alpha_i, \quad Z_{a,i} Z_{a,i}^* \leq \alpha_i && \text{for all } a, i = 1, \ldots, m-1 \\
& [M_{a|x}, N_{b|y}] = [M_{a|x}, Z_{b,i}^{(*)}] = [N_{b|y}, Z_{a,i}^{(*)}] = 0 && \text{for all } a, b, x, y, i
\end{aligned} \tag{7}$$

The only small difference here is that we have included the simplification that the initial state on Alice, Bob and Eve's system is pure.

Tips/tricks/modifications

- You may have noticed that we do not use the m^{th} w_i and t_i. One can include these in the optimization by allowing the summation to be from $i = 1$ to m as opposed to $m - 1$ (this also includes modifying the summation defining the constant c_m). You may find better rates doing this (it can't be worse). However, the SDP can become unstable so be careful to check the solution status if you are keeping the final term in the sum. In the example code this is an option toggled by the constant KEEP_M.

- To get much faster computations you can commute the outer summation and the infimum in (7), i.e., compute

$$\sum_{i=1}^{m-1} \frac{w_i}{t_i \ln 2} \inf \sum_a \langle\psi| \ldots |\psi\rangle. \tag{8}$$

  This can only decrease the value of the optimization and hence it is sufficient for the purpose of lower bounding rates. The main advantage gained here is that it reduces the number of variables in the NPA relaxations. Rather than running a single SDP with a $Z_{a,i}$ variable for each pair (a, i), we can instead run $m - 1$ much smaller SDPs with a $Z_{a,i}$ variable for each a only. This significantly reduces the overall runtime as it now scales linearly with the number of nodes in the Gauss-Radau quadrature. We applied this everywhere in our computations. However, a user should be aware that there is no guarantee that this does not lead to a significant loss of rate.

- We can assume that all of Alice and Bob's measurements in the problem are projective. I.e., we can include operator constraints like $M_{a|x}^2 = M_{a|x}$.

- If we're looking at QKD rates with noisy preprocessing we can incorporate this by modifying the objective. If q is the probability that Alice flips her raw key bit then in the objective function we make the replacement

$$M_{a|x^*} \mapsto (1-q) M_{a|x^*} + q M_{a \oplus 1|x^*}.$$

- If we are implementing noisy preprocessing and KEEP_M = 0 (we do not compute the m^{th} term) then we can replace c_m with

$$c_m = 2q(1-q) + \sum_{i=1}^{m-1} \frac{w_i}{t_i \ln 2}.$$

  Effectively, when KEEP_M = 0 we bound the final term in a trivial way. If we have noisy preprocessing then we can bound it in a slightly less trivial way.

3 Example scripts

In the github repository DI-rates there are several example scripts to help with getting started. This includes:

- chsh_local.py – Bounds $H(A|X=0,Q_E)$ for devices constrained by a CHSH score. Can be usedd to numerically recover the known tight analytical bound (see Figure 1 in arXiv:2016.13692).
- 2222_global.py – Bounds $H(AB|X=0,Y=0,Q_E)$ for devices constrained by some full distribution (see Figure 3 in arXiv:2016.13692).
- qkd.py – Bounds $H(A|X=0,Q_E) - H(A|X=0,Y=2,B)$ for devices constrained by some full distribution (see Figure 4 in arXiv:2016.13692).

3.1 ncpol2sdpa

The scripts use the python package ncpol2sdpa in order to generate the SDP relaxations of the NCPOP. Fairly detailed documentation for ncpol2sdpa is available online. You should use the version of ncpol2sdpa available on my github fork. You can install it using pip via the command

```
pip install git+https://github.com/peterjbrown519/ncpol2sdpa.git
```

Example script: qkd.py

Let us run through some of the aspects of the script related to the ncpol2sdpa package. Ncpol2sdpa allows a user to define an arbitrary NCPOP and it will convert it to an SDP and solve the resulting SDP.

When we begin using ncpol2sdpa we first define the monomials that generate the NCPOP. For the measurements of Alice and Bob we define some Hermitian operators and we also define Eve's nonhermitian operators. When generating measurement operators ncpol2sdpa will not generate the final operator of each measurement, you can instead define it via $M_n = I - \sum_{i=1}^{n-1} M_i$ for an n-outcome measurement.

```
# number of outputs for each inputs of Alice / Bobs devices
# (Dont need to include 3rd input for Bob here as we only constrain the statistics
# for the other inputs).
A_config = [2,2]
B_config = [2,2]

# Operators in problem
A = [Ai for Ai in ncp.generate_measurements(A_config, 'A')]
B = [Bj for Bj in ncp.generate_measurements(B_config, 'B')]
Z = ncp.generate_operators('Z', 2, hermitian=0)
```

Here we created a list of lists for Alice and Bob's measurements. I.e., the measurement operator $M_{0|x}$ is the object A[x][0]. The actual objects used to represent the monomials are operators from sympy. You can really treat these objects as generators of some *-algebra. You can multiply them, add them or even apply the involution operation (Hermitian-conjugate/Dagger).

For instance we can use these operators to define our objective function (using the simplification mentioned in (8))

```
def objective(ti, q):
    """
    Returns the objective function for the faster computations.
        Key generation on X=0
        Only two outcomes for Alice

        ti      --    i-th node
        q       --    bit flip probability
    """
    obj = 0.0
    F = [A[0][0], 1 - A[0][0]]                # POVM for Alices key gen measurement
```

```
    for a in range(A_config[0]):
        b = (a + 1) % 2                          # (a + 1 mod 2)
        M = (1-q) * F[a] + q * F[b]              # Noisy preprocessing povm element
        obj += M * (Z[a] + Dagger(Z[a]) + (1-ti)*Dagger(Z[a])*Z[a]) + ti*Z[a]*Dagger(Z[a])

    return obj
```

This function defines the noisy-preprocessing objective function for each node t_i.

We can also use our monomials to define the moment equalities for a given distribution. We do this in the following function

```
def score_constraints(sys, eta=1.0):
    """
    Returns the moment equality constraints for the distribution specified by the
    system sys and the detection efficiency eta. We only look at constraints coming
    from the inputs 0/1. Potential to improve by adding input 2 also?

        sys     --      system parameters
        eta     --      detection efficiency
    """

    # Extract the system
    [id, sx, sy, sz] = [qtp.qeye(2), qtp.sigmax(), qtp.sigmay(), qtp.sigmaz()]
    [theta, a0, a1, b0, b1, b2] = sys[:]
    rho = (cos(theta)*qtp.ket('00') + sin(theta)*qtp.ket('11')).proj()

    # Define the first projectors for each of the measurements of Alice and Bob
    a00 = 0.5*(id + cos(a0)*sz + sin(a0)*sx)
    a01 = id - a00
    a10 = 0.5*(id + cos(a1)*sz + sin(a1)*sx)
    a11 = id - a10
    b00 = 0.5*(id + cos(b0)*sz + sin(b0)*sx)
    b01 = id - b00
    b10 = 0.5*(id + cos(b1)*sz + sin(b1)*sx)
    b11 = id - b10

    A_meas = [[a00, a01], [a10, a11]]
    B_meas = [[b00, b01], [b10, b11]]

    constraints = []

    # Add constraints for p(00|xy)
    for x in range(2):
        for y in range(2):
            constraints += [A[x][0]*B[y][0] - (eta**2 * (rho*qtp.tensor(A_meas[x][0], B_meas
    [y][0])).tr().real + \
                        + eta*(1-eta)*((rho*qtp.tensor(A_meas[x][0], id)).tr().real + (rho*
    qtp.tensor(id, B_meas[y][0])).tr().real) + \
                        + (1-eta)*(1-eta))]

    # Now add marginal constraints p(0|x) and p(0|y)
    constraints += [A[0][0] - eta * (rho*qtp.tensor(A_meas[0][0], id)).tr().real - (1-eta)]
    constraints += [B[0][0] - eta * (rho*qtp.tensor(id, B_meas[0][0])).tr().real - (1-eta)]
    constraints += [A[1][0] - eta * (rho*qtp.tensor(A_meas[1][0], id)).tr().real - (1-eta)]
    constraints += [B[1][0] - eta * (rho*qtp.tensor(id, B_meas[1][0])).tr().real - (1-eta)]

    return constraints[:]
```

Given a description of a two-qubit system "sys" and a detection efficiency "eta" the function defines a list of constraints of the form

$$\mathrm{Tr}\left[\rho M_{a|x} N_{b|y}\right] - p(a,b|x,y) = 0$$

as well as marginal constraints.

We can also define the constraints for projective measurements and commutation relations.

```
def get_subs():
    """
    Returns any substitution rules to use with ncpol2sdpa. E.g. projections and
    commutation relations.
    """
    subs = {}
    # Get Alice and Bob's projective measurement constraints
    subs.update(ncp.projective_measurement_constraints(A,B))

    # Finally we note that Alice and Bob's operators should All commute with Eve's ops
    for a in ncp.flatten([A,B]):
        for z in Z:
            subs.update({z*a : a*z, Dagger(z)*a : a*Dagger(z)})

    return subs
```

Whenever we have an operator equality constraint of the form $X = Y$ where X and Y are monomials, ncpol2sdpa allows us to define a substitution rule. That is, we would define a rule $X \mapsto Y$. Then whenever ncpol2sdpa encounters the monomial X, it will replace it with Y. This allows ncpol2sdpa to reduce the size of the SDP and speed up the computations. Here we defined substitution rules for the projective measurement constraints and the commutation relations.

We can also define intermediate relaxations. I.e., we have an SDP relaxation that includes all monomials of size 2 but also include some select monomials of size 3 or larger. To do this, we just need to define a list of the additional monomials that we want to include. In the script this is defined in the following function.

```
def get_extra_monomials():
    """
    Returns additional monomials to add to sdp relaxation.
    """

    monos = []

    # Add ABZ
    ZZ = Z + [Dagger(z) for z in Z]
    Aflat = ncp.flatten(A)
    Bflat = ncp.flatten(B)
    for a in Aflat:
        for b in Bflat:
            for z in ZZ:
                monos += [a*b*z]

    # Add monos appearing in objective function
    for z in Z:
        monos += [A[0][0]*Dagger(z)*z]

    return monos[:]
```

In this function we define monomials of the form $M_{a|x}N_{b|y}Z_c^{(*)}$ and $M_{0|0}Z_c^*Z_c$. The former seem to improve the rates of the problem and the latter are important as they appear in the objective function. The monomials you choose here are completely flexible, it's a tradeoff between speed and convergence – worth playing around with this.

Now that we have our various functions that help define the problem, we can finally ask ncpol2sdpa to generate an SDP relaxation.

```
substitutions = get_subs()              # substitutions used in ncpol2sdpa
moment_ineqs = []                       # moment inequalities
moment_eqs = []                         # moment equalities
op_eqs = []                             # operator equalities
op_ineqs = []                           # operator inequalities
extra_monos = get_extra_monomials()     # extra monomials

# Defining the test sys
```

```
test_sys = [pi/4, 0, pi/2, pi/4, -pi/4, 0]
test_eta = 0.99
test_q = 0.01


ops = ncp.flatten([A,B,Z])          # Base monomials involved in problem
obj = objective(1,test_q)     # Placeholder objective function


sdp = ncp.SdpRelaxation(ops, verbose = VERBOSE-1, normalized=True, parallel=0)
sdp.get_relaxation(level = LEVEL,
                       equalities = op_eqs[:],
                       inequalities = op_ineqs[:],
                       momentequalities = moment_eqs[:] + score_constraints(test_sys, test_eta)
    ,
                       momentinequalities = moment_ineqs[:],
                       objective = obj,
                       substitutions = substitutions,
                       extramonomials = extra_monos)
```

In the script LEVEL is the level of the NPA relaxation – e.g., for LEVEL=2 we include all monomials of length at most 2. We first initiate the relaxation object in line 582 where we provide it with the generating set of monomials (Alice's POVM elements / Bob's POVM elements / Eve's Z operators). Then we can request a relaxation of our problem by defining the relaxation level, constraints (operator and moment), objective function (always minimizes), substitution rules, and any additional monomials we want to add to the relaxation.

We can solve the resulting SDP using sdp.solve() or we can change the constraints of the problem using sdp.process_constraints() – see the ncpol2sdpa documentation for further details on these methods.

To compute the lower bound on $H(A|X=0,Q_E)$ for our test system there is a function compute_entropy() provided with the code.

```
def compute_entropy(SDP, q):
    """
    Computes lower bound on H(A|X=0,E) using the fast (but less tight) method

        SDP   --   sdp relaxation object
        q     --   probability of bitflip
    """
    ck = 0.0         # kth coefficient
    ent = 0.0         # lower bound on H(A|X=0,E)

    # We can also decide whether to perform the final optimization in the sequence
    # or bound it trivially. Best to keep it unless running into numerical problems
    # with it. Added a nontrivial bound when removing the final term
    # (WARNING: proof is not yet in the associated paper).
    if KEEP_M:
        num_opt = len(T)
    else:
        num_opt = len(T) - 1
        ent = 2 * q * (1-q) * W[-1] / log(2)

    for k in range(num_opt):
        ck = W[k]/(T[k] * log(2))

        # Get the k-th objective function
        new_objective = objective(T[k], q)

        SDP.set_objective(new_objective)
        SDP.solve('mosek', solverparameters = {'num_threads': int(NUM_SUB_WORKERS)})

        if SDP.status == 'optimal':
            # 1 contributes to the constant term
            ent += ck * (1 + SDP.dual)
        else:
```

```
            # If we didn't solve the SDP well enough then just bound the entropy
            # trivially
            ent = 0
            if VERBOSE:
                print('Bad solve: ', k, SDP.status)
            break

    return ent
```

To compute the DIQKD rate there is also a function compute_rate() which will take care of the $H(A|X = 0, Y = 2, B)$ term. There is also a function optimise_rate() which when given a starting two-qubit system, preprocessing parameter q and detection efficiency η will try to optimize the choice of two-qubit system and q to maximize the rate. This meta-optimization is done using ideas similar to those presented in [2].

References

[1] S. Pironio, M. Navascués, and A. Acín, "Convergent relaxations of polynomial optimization problems with noncommuting variables," *SIAM Journal on Optimization*, vol. 20, no. 5, pp. 2157–2180, 2010.

[2] S. M. Assad, O. Thearle, and P. K. Lam, "Maximizing device-independent randomness from a Bell experiment by optimizing the measurement settings," *Physical Review A*, vol. 94, no. 1, p. 012304, 2016.