Convex Optimization for problems in Quantum Information Theory

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Aryaman Jeendgar
ID No. 2019B5AA0767H

Under the supervision of:

Dr. Riley Murray
&
Dr. Sarmistha Banik

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS
December 2023

Declaration of Authorship

I, Aryaman Jeendgar, declare that this Undergraduate Thesis titled, 'Convex Optimization for problems in Quantum Information Theory' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Certificate

This is to certify that the thesis entitled, "*Convex Optimization for problems in Quantum Information Theory*" and submitted by Aryaman Jeendgar ID No. 2019B5AA0767H in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor	*Co-Supervisor*
Dr. Riley Murray	Dr. Sarmistha Banik
Senior Member of Technical Staff,	Professor,
Sandia National Laboratories	BITS-Pilani Hyderabad Campus
Date:	Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS

Abstract

Master of Science(Hons.) Physics

Convex Optimization for problems in Quantum Information Theory

by Aryaman Jeendgar

Convex optimization is an important tool in Quantum Information (QI). CVXPY is used in QI-focused Python packages like: QuTiP, qiskit-optimization, Rigetti's Grove, and Sandia National Labs' pyGSTi [17, 21]. The goal of this thesis began from working towards the inclusion and implementation of various functions and constraints highly relevant for applications in QI such as von Neumann entropy, quantum relative entropy, and the operator relative entropy cone. The scope of this work eventually proliferated to working on other aspects such as the implementation of a high-level module for verifying the KKT conditions (briefly discussed in this thesis). In summary, the core contribution of the work done as a part of this thesis is the implementation of semidefinite representations of the above-stated functions described in [11] within the CVXPY modeling platform.

Acknowledgements

This thesis summarizes a great deal of the work that I have done towards extending CVXPY over my almost (at the time of this writing) year and a half association with the CVXPY group and the community at large. None of it would've been possible in its current form without the constant support and input from my advisor, Dr. Riley Murray. From working on the `FiniteSet` PR when I first touched base with the CVXPY codebase to now extending the work we began back during GSoC-22, I am *deeply* grateful to Dr. Riley for *always* being around, as an advisor, mentor and friend.

At the same time I would be amissed to not thank the rest of the members of the CVXPY group including (but not limited to), Steven Diamond, Philipp Schiele and Akshay Agrawal for all of their mentorship and help throughout this time (and to more beyond)!

I would also like to thank My Co-Supervisor, Prof. Sarmistha Banik. From being an instructor beyond par in the courses I took under her to introducing me to the exciting world of Neutron Star physics, which we worked on for two-semesters straight to also being an exceptional mentor. I am truly grateful to have been associated with Prof. Banik during my stay at BITS. Speaking of which I'd like to thank the entire Physics department at BITS Hyderabad for giving me a cutting-edge education in the physical sciences, much of which will be indispensable as I slowly progress towards becoming an independent researcher. Thank you!

Back home, I'd like to thank my parents, my younger brother and my closest friends for being my anchor in the most challenging of times (of which, there were many :)).

Contents

List of Figures

Abbreviations/Notations

DSL	**D**omain **S**pecific **L**anguage
DCP	**D**isciplined **C**onvex **P**rogramming
`QRE`	`Q`uantum `R`elative `E`ntropy
KKT	**K**arush-**K**uhn-**T**ucker
`OREC`	`O`perator `R`elative `E`ntropy `C`one
`sREC`	`s`calar `R`elative `E`ntropy `C`one
NCP	**N**on-**C**ommutative **P**erspective
WMGM	**W**eighted **M**atrix **G**eometric **M**ean
rSOC	**r**otated **S**econd **O**rder **C**one
`VNE`	`V`on-`N`eumann `E`ntropy
$[p]$ for integer p	$\{1, 2, \ldots, p\}$

Dedicated to the Indomitable Human Spirit...

Chapter 1

A Generic introduction to Convex & Conic Optimization

This chapter introduces some basic notions that will be useful for understanding the work done throughout the entire thesis. Much of this is classical content and borrows heavily from [4, 3, 5, 13] and [14].

1.1 Whetting the appetite, *basic*

1.1.1 What is mathematical optimization?

Mathematical optimization deals with constrained programs taking on the following generic form:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq 0, \quad i = 1 \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{aligned}$$

Where, $x \in \mathbb{R}^n$

and includes the following general areas [4]:

1. *Modelling*: Techniques for writing out applied problems in the above general format
2. *Optimization Theory*: Characterizing the nature of the optima of various subclasses of the above problem

3. *Optimization Methods*: Development and analysis of algorithms for solving the above programs

4. *Software*: Development of higher-level and easy-to-use software for posing and solving various kinds of optimization problems

In the above, f_0 is the *objective-function* that we are trying to minimize (or maximize, depending on the particular formulation), the f_i are the so-called *inequality-functions* of the problem (since they are used to describe the sets that the optimal solution ought to lie in via an inequality) and the h_i are the functions used to describe equality constraints in the problem.

A point is said to be *feasible* if it satisfies each of the constraints in the problem i.e.:

Definition 1. The point x for an optimization problem is feasible if:

$$\begin{aligned} &x \in \mathcal{D} \\ &f_i(x) \leq 0 \quad \forall i \in [m] \\ &h_i(x) = 0 \quad \forall i \in [p] \end{aligned}$$

1.1.2 Optimization software

Much of our work in this thesis is dedicated to extending the `CVXPY`, [8], modeling platform with functionality for being able to model quantum information problems.

`CVXPY` is a *domain-specific-language* that provides a convenient algebra (via it's various functions and constraint sets implemented as `Atoms` and `Constraint` classes) for expressing optimization problems within python itself. `CVX`, [15] and `JuMP`, [9], serve similar roles within Matlab and Julia. As an example, note the following linear program and its CVXPY implementation.

$$\begin{aligned} \text{minimize} \quad & 5x_1 + x_2 + 2x_3 \\ \text{subject to} \quad & 3x_1 + x_2 + 0.5x_3 \geq 6 \\ & 3x_1 + 2x_2 + 4x_3 \geq 15 \\ & 2x_1 + x_3 \geq 5 \\ & x_1 + 4x_2 \geq 7 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

```
import numpy as np
import cvxpy as cp

x = cp.Variable(3)
obj = cp.Minimize(5 * x[0] + x[1] + 2 * x[2])
cons = [
    3 * x[0] + x[1] + 0.5 * x[2] >= 6,
    3 * x[0] + 2 * x[1] + 4 * x[2] >= 15,
    2 * x[0] + x[2] >= 5,
    x[0] + 4 * x[1] >= 7,
    x >= 0
]

prob = cp.Problem(obj, cons)
prob.solve()
```

FIGURE 1.1: Linear program, single decision variable with it's CVXPY demo

This role is distinct from optimization solvers such as `MOSEK`, `GUROBI`, `CLARABEL` (among others), which are '*lower-level*' software programs (compared to their DSL counterparts). In essence, DSL's like CVXPY, CVX and JuMP take in the specification of an optimization problem in the symbolic form shown above, ensure that the problem has been constructed in accordance with Disciplined Convex Programming (DCP), [16], rules, and then convert this algebraic description of the problem into a representation that these lower-level solvers can accept and work with — the output of which is then passed back through the chain and the user's symbolic description is then populated with different values of the solution. We will briefly get a chance to work with the lower-level part of this pipeline in Chapter 2.

We end this section by implementing a stylistic problem in CVXPY which has *helper*-variables that don't directly contribute to the construction of the objective function. In this particular construction, the *auxiliary* variable z that has been introduced into the equality constraint can be thought of as representing the *slack* between the value of the constraint function on the LHS and the RHS of the inequality **AT** the optimal value of the problem.

$$
\begin{aligned}
\text{minimize} \quad & 3x_1 + 2x_2 \\
\text{subject to} \quad & x_1 + x_2 \geq 5 \\
& 2x_1 - 3x_2 + z = 1 \\
& \log \sum_{i=1}^{2} e^{x_i} \leq 10 \\
& x_1 + 4x_2 \geq 7 \\
& z \geq 0
\end{aligned}
$$

```python
import numpy as np
import cvxpy as cp

x = cp.Variable(2)
z = cp.Variable()
obj = cp.Minimize(3 * x[0] + 2 * x[1])
cons = [
    x[0] + x[1] >= 5,
    2 * x[0] - 3 * x[1] + z == 1,
    cp.log_sum_exp(x) <= 10,
    z >= 0
]

prob = cp.Problem(obj, cons)
prob.solve()
```

FIGURE 1.2: Convex optimization problem with an *auxiliary*, slack variable

1.2 Some fundamental theory, *intermediate*

1.2.1 Basic definitions

In this section, we do a whirlwind overview of some definitions that are fundamental to much of the discussion ahead [13].

Definition 2. A set $\mathcal{D}$ is a convex set if it completely contains the line segment between any two points in the set $\mathcal{D}$

$$\forall x, y \in \mathcal{D}, 0 \leq t \leq 1 \Rightarrow tx + (1-t)y \in \mathcal{D}$$

Definition 3. A function $f(\cdot)$ with domain $\mathcal{D}$ is convex if:

$$f\left(\alpha x + (1-\alpha)y\right) \leq \alpha f(x) + (1-\alpha)f(y)$$

Definition 4. An alternate characterization of convexity for a function is by studying its epigraph, which is a set that can be associated with every function f. The *epigraph* of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the set:

$$\mathbf{epi}(f) := \{(x,t) | x \in \mathbf{dom}(f), f(x) \leq t\}$$

The above definition implies that minimizing over the epigraph is equivalent to minimizing $f(x)$. Furthermore, a function f is convex if and only if its epigraph is convex.

$$\begin{aligned} &\text{minimize} && t \\ &\text{subject to} && f(x) \leq t \end{aligned}$$

With the above in place, we can give the following definition for a *convex-optimization* problem:

Definition 5. A convex optimization problem is of the form:

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && f_i(x) \leq 0, \quad i \in [m] \\ & && Ax = b \end{aligned}$$

Where the functions $f(\cdot)$ and $f_i(\cdot) \forall i$ are all convex functions and the equality constraints are affine functions. This ensures that the feasible set of the problem is a convex set.

Now, note that the two problems that were implemented in the previous section are both convex optimization programs! To see this, note that the objective functions of both of the problems are linear in their respective optimization variables, which combined with the convexity of linear functions, means that the objectives of both problems are indeed convex.

Similarly, all of the constraint functions, f_i of the first program, are linear in nature and, hence, also lead to a convex feasible region.

For the second problem, the objective and constraints [1, 2, 4, 5] are linear and hence again, convex. The third constraint makes use of the `log_sum_exp` atom, which means we have a convex function as a constraint function, with an appropriately directed inequality. Hence, since each of the constraints are individually convex, the final feasible region for the problem which is the intersection of each of them is also convex, making the problem convex as a whole!

Definition 6. The cone of positive definite matrices can be characterized in these the following two equivalent ways.

$$\begin{aligned}\mathcal{S}^n_{++} &= \left\{X \in \mathcal{S}^n | z^T X z > 0, \forall z \in \mathbb{R}^n\right\} \\ &= \{X \in \mathcal{S}^n | \lambda_i(X) > 0, i \in [n]\}\end{aligned}$$

PD-ness is symbolically depicted as $X \succ 0$ similarly, PSD-ness by $X \succeq 0$

1.2.2 A toy problem using the `QRE`

We shall delve into a more thorough survey of the kinds of problems that can be implemented using the `QRE` (one of the major contributions to come out of this work) in Chapter 7. Here, we provide a stylized example of the computation of the nearest correlation matrix in the sense of the `QRE`.

A *correlation matrix* is a symmetric positive definite matrix with a unit diagonal. The nearest correlation matrix problem attempts to approximate a given symmetric matrix A with a matrix X that is constrained to be a correlation matrix (see [2] for more information), . This is typically done in the sense of the Frobenius norm, but here for demonstration purposes, we do it with the `QRE`.

$$\begin{aligned}\text{minimize} \quad & ||A - X||_{\texttt{QRE}} \\ \text{subject to} \quad & \text{diag}(X) = e \\ & X \succeq 0\end{aligned}$$

```
import cvxpy as cp
import numpy as np
n = 4
np.random.seed(0)
A = np.random.randn(n, n)
A = M @ M.T
X = cp.Variable(shape=(n, n), symmetric=True)
obj = cp.Minimize(cp.quantum_rel_entr(M, X))
cons = [
    cp.diag(X) == np.ones((n,)),
    X >> 0
]
prob = cp.Problem(obj, cons)
prob.solve()
```

FIGURE 1.3: Nearest Correlation matrix in the sense of the quantum relative entropy

1.3 On Conic Optimization, *Advanced*

1.3.1 Convex cones and the Conic reformulation

In this section we briefly introduce the conic formulation of convex optimization. Much of the content here follows the excellent review, [14]

Definition 7. A convex cone is a set that is both:

1. Closed under scalar multiplication, i.e. $x \in \mathcal{C} \Rightarrow \lambda x \in \mathcal{C} \forall \lambda \in \mathbb{R}_+$
2. Closed under addition, i.e. $x, y \in \mathcal{C} \Rightarrow x + y \in \mathcal{C}$

Definition 8. A cone $\mathcal{C}$ is *pointed* if and only if $\mathcal{C} \cap -\mathcal{C} = \{0\}$, where $-\mathcal{C}$ stands for the set $\{x| - x \in \mathcal{C}\}$

Definition 9. A cone $\mathcal{C}$ is *solid* if and only if $\mathbf{int}\,\mathcal{C}$ is nonempty

With the above in mind, we can write the conic optimization problem as:

Definition 10. Let $\mathcal{C} \subseteq \mathbb{R}^n$ be a pointed, solid closed convex cone. The primal conic optimization problem as:

$$\inf_{x \in \mathbb{R}^n} \quad c^T x \text{ s.t. } Ax = b \text{ and } x \in \mathcal{C}$$

Where $x \in \mathbb{R}^n$ is the column vector that is being optimized and the problem data is given by cone $\mathcal{C}$, a $m \times n$ matrix A and two column vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$.

The conic optimization problem can hence be viewed as minimizing a linear function over the intersection of a convex cone and an affine subspace.

For example, the linear optimization problem can be formulated by choosing the cone $\mathcal{C}$ to be the positive orthant $\mathbb{R}^n_+$, leading to the below standard formulation:

$$\inf_{x \in \mathbb{R}^n} \quad c^T x \text{ s.t. } Ax = b \text{ and } x \geq 0$$

The major difference between the above-defined conic problem and the standard convex problem lies in the special choice of the feasible set $\mathcal{S} = \mathcal{C} \cap \mathcal{L}$, where $\mathcal{L}$ is the affine subspace defined by $\mathcal{L}\{x \in \mathbb{R}^n | Ax = b\}$. Since $\mathcal{C}$ and $\mathcal{L}$ are convex, then $\mathcal{C} \cap \mathcal{L}$ is also convex ensuring the primal conic optimization problem is also a convex optimization problem.

Most importantly, though, is the result that the above formulation of the convex optimization problem in terms of convex cones is just as general as its traditional convex counterpart!

The principal advantage of the conic formulation of a convex program, however, lies in the very symmetrical formulation of its corresponding dual problem, which is discussed in the next section.

1.3.2 Conic duality

Each primal convex optimization problem admits a corresponding, strongly related dual problem that may be computed using the theory of Lagrange duality. However, dual problems computed this way are far from symmetric *w.r.t* the structure of the primal.

The Lagrangian dual of the conic problem stated above, however, can be expressed in a very symmetric form with its original primal, in conic form by exploiting the notion of a dual cone.

Definition 11. The dual of a cone $\mathcal{C} \subseteq \mathbb{R}^n$ is defined by

$$\mathcal{C}^* = \left\{x^* \in \mathbb{R}^n | x^T x^* \geq 0 \forall x \in \mathcal{C}\right\}$$

Furthermore, we have the following key results

Theorem 1. *If $\mathcal{C}$ is a closed convex cone, its dual $\mathcal{C}^*$ is another closed convex cone. The dual $(\mathcal{C}^*)^*$ of $\mathcal{C}^*$ is equal to $\mathcal{C}$*

Theorem 2. *If $\mathcal{C}$ is a solid, pointed, closed convex cone, its dual $\mathcal{C}^*$ is another solid, pointed, closed convex cone and $(\mathcal{C}^*)^* = \mathcal{C}$*

We are now ready to state the dual conic problem

Definition 12. The Lagrangian dual of the primal conic problem is defined by:

$$\sup_{y \in \mathbb{R}^m, s \in \mathbb{R}^n} \quad b^T y \text{ s.t. } A^T y + s = c \text{ and } s \in \mathcal{C}^*$$

Where $y \in \mathbb{R}^m$ and $s \in \mathbb{R}^n$ are the column vectors that are being optimized over.

Clearly, the above dual problem has an identical structure to the primal conic problem, in that it also involves the optimization of a linear function over the intersection of a convex cone and an affine subspace.

One of the major advantages of having access to a dual problem (for a convex optimization problem) that is readily analyzable is the convenience of being able to verify the *KKT conditions*, which are optimality conditions for convex-constrained programming.

1.3.3 Optimality for Conic Problems

Before proceeding any further, we first require the notions of Strong and Weak duality (note that p^*, d^* are the optimal values of the primal and dual problems respectively)

Weak duality states that the optimium value of the dual is always a lower-bound on the optimal value of the primal, i.e. $d^* \leq p^*$. Since weak duality *always* holds, if solving the dual for a given problem is easier than the primal, then it can be used to obtain nontrivial lower bounds for the same.

Strong duality states, $d^* = p^*$. This does *not* hold in general. Conditions that guarantee that strong duality hold, are called *constraint qualifications*, the most significant of which are due to Slater.

Slater's condition relates to the existence of Lagrange multipliers for a convex program and guarantees the strong duality. For conic programs, it can simply be stated as

$$\exists x \in \mathbf{int}(C) \text{ s.t. } Ax = b$$

Theorem 3 (KKT conditions for conic problems)**.** *The conic problem:*

$$p^* = \inf_{x \in \mathbb{R}^n} \quad c^T x \ \textit{s.t.}\ Ax = b \ \textit{and}\ x \in \mathcal{C}$$

admits the dual bound $p^* \geq d^*$, where

$$d^* = \sup_{y \in \mathbb{R}^m, s \in \mathbb{R}^n} \quad b^T y \ \textit{s.t.}\ A^T y + s = c \ \textit{and}\ s \in \mathcal{C}^*$$

If both problems are strictly feasible, then the duality gap is zero: $p^* = d^*$ and both value are attained. Then a pair (x, y) is primal-dual optimal if and only if the KKT conditions are satisfied, i.e.

1. *Primal feasibility: $x \in \mathcal{C}, Ax = b$*
2. *Dual feasibility: $c - A^T y \in \mathcal{C}^*$*
3. *Complementary slackness: $(c - A^T y)^T x = 0$*

The reason the stationarity condition is not seen in the above statement is because $\mathcal{L}(x, \lambda, y) = c^T x + y^T(b - Ax) - \lambda^T x$, and $\nabla_x \mathcal{L}(x, \lambda, y) = 0 \Rightarrow (c - A^T y - \lambda) = 0$, and λ can be easily eliminated from the resultant set of equations.

Chapter 2

Dual variables for the n-dimensional power cone in CVXPY

2.1 What is the Power Cone?

Definition 13. The general n-dimensional power cone with a "long left-hand side" may be defined as follows, [2]:

$$\mathcal{P}_n^{\alpha_1,\ldots,\alpha_m} = \left\{ x \in \mathbb{R}^n : \Pi_{i=1}^m x_i^{\alpha_i} \geq \sqrt{\sum_{i=m+1}^{n} x_i^2}, x_1, \ldots, x_m \geq 0 \right\}$$

Where, $\sum_i \alpha_i = 1$ (i.e. the α_i's are elements of the n -dimensional simplex)

However, CVXPY defines the power cone slightly differently (it is this definition that we work with for the remnant of our discussion):

Definition 14.

$$\mathcal{P}_n^{\alpha_1,\ldots,\alpha_n} = \left\{ x \in \mathbb{R}^n : \Pi_{i=1}^{n-1} x_i^{\alpha_i} \geq |x_n| \right\}$$

i.e. we impose $m = n - 1$. The above will be referred to as `PowConeND` (typeset `as such`) henceforth.

2.2 The dual of the Power Cone

Lemma 1. *The dual cone to* `PowConeND` *is[2]:*

$$(\mathcal{P}^{\alpha_1,\ldots,\alpha_m})_n^* = \left\{ y \in \mathbb{R}^n : \left(\frac{y_1}{\alpha_1}, \ldots, \frac{y_{n-1}}{\alpha_{n-1}}, y_n \right) \in \mathcal{P}^{\alpha_1,\ldots,\alpha_m} \right\}$$

Where $(y_1, \ldots, y_n)$ are the corresponding dual variables.

2.3 A *simpler* representation of the N-dimensional power cone

Canonicalization is the process of describing a set (in our cases, convex sets) in terms of other, more 'fundamental' sets — in the case of CVXPY, as mentioned prior, these 'fundamental' sets will be the convex cones which the solvers that CVXPY has an existing solver interface with, support.

It is this process that allows a user's specification of a higher-level program written in CVXPY by leveraging its various supported `Atom` and `Constraint` classes can communicate with the output of the lower-level solvers that CVXPY calls.

For our convex cone of interest, namely `PowConeND`, this canonicalization is particularly intuitive, and we can, in-fact, represent a singular `PowConeND` constraint as the intersection of multiple 3-dimensional power cones (henceforth referred to as `PowCone3D`). This is achieved by recursively splitting `PowConeND` down into `PowCone3D` constraints like so, [1]:

$$
\begin{aligned}
& x_1^{\alpha_1} t_1^{\gamma_1} \geq |x_n| \quad \text{where, } \gamma_1 = \alpha_2 + \ldots + \alpha_{n-1} \\
& \ldots \\
& x_i^{\frac{\alpha_i}{\Pi_{j=1}^{i-1}\gamma_j}} t_i^{\gamma_i} \geq t_{i-1} \quad \text{where, } \gamma_i = \frac{\sum_{j=i+1}^{n-1} \alpha_j}{\Pi_{i=1}^{i-1}\gamma_j} \qquad \forall i = 2, \ldots, n-3 \\
& \ldots \\
& x_{m-1}^{\frac{\alpha_{m-1}}{\Pi_{j=1}^{n-3}\gamma_j}} x_m^{\frac{\alpha_m}{\Pi_{j=1}^{n-3}\gamma_j}} \geq t_{n-3}
\end{aligned}
$$

Essentially, every N -dimensional `PowConeND` constraint can be represented by $(N-2)$ `PowCone3D` constraints.

2.4 Dual variables corresponding to this representation

The dual variables for each of these `PowCone3D` subcones can be given as follows, [1]:

$$\left(\frac{y_1}{\alpha_1}\right)\left(\frac{R_1}{\gamma_1}\right) \geq |y_n|$$

$$\dots$$

$$\left(\frac{y_i \times \Pi_{i=1}^{i-1}\gamma_j}{\alpha_i}\right)^{\frac{\alpha_i}{\Pi_{j=1}^{i-1}\gamma_j}} \left(\frac{R_i}{\gamma_i}\right) \geq R_{i-1} \qquad \forall i = 2, \dots, n-3$$

$$\dots$$

$$\left(\frac{y_{n-2} \times \Pi_{j=1}^{n-3}\gamma_j}{\alpha_{n-2}}^{\frac{\alpha_{n-2}}{\Pi_{j=1}^{n-3}\gamma_j}}\right) \left(\frac{y_m \times \Pi_{j=1}^{n-3}\gamma_j}{\alpha_{n-1}}\right)^{\frac{\alpha_m}{\Pi_{j=1}^{n-3}\gamma_j}} \geq R_{n-3}$$

where the y 's are the duals to the x 's in the original problem and R 's are the duals to the auxiliary variables t.

2.5 Implementing all of the above within CVXPY

Implementing the above recipe within CVXPY was an interesting exercise in hacking the libraries' (undocumented) internals. Before my PR for adding dual variable support for `PowConeND`, CVXPY supported dual variable recovery for *every* supported fundamental cone — the convex cones for which this feature had not been implemented were the ones that required the implementation of an explicit canonicalization procedure at the level of CVXPY's internals (before the problem's representation got passed down to lower-level solvers) — there's three such cones in CVXPY, `PowConeND, RelEntrConeQuad` and `OpRelEntrConeQuad`.

As of today, there does not exist a closed form expression for the dual variables of the latter two cones (an expression for the `sREC`, which is the cone that the `RelEntrConeQuad` is approximating, may be readily derived by exploiting its trivial relation to the exponential cone — this is explored in Chapter5).

Whereas, the dual cone for the full Operator Relative Entropy Cone is an open problem (for both the exact `OREC` and also `OpRelEntrConeQuad`)

We spent a considerable amount of time studying the exact way dual variables are passed around while CVXPY constructs the `SolvingChain` for solving a user-defined problem. After much investigation, we honed in on the `invert::Canonicalization` methods as the exact place for implementing the above-defined math for recovering the dual variables for a `PowConeND` constraint from it's constituent `PowCone3D` constraints.

In the case of `PowConeND` constraint, it has a dedicated canonicalization class, `Exotic2Common subclasses Canonicalization`, namely, we overrode the `invert::Exotic2Common` method, where we implemented the above math for choosing the correct dual variables for the `PowConeND` constraint corresponding to the generated $(N-2)$ `PowCone3D` constraints.

```python
class PowConeND(Cone):

    # ... omitted code ...

    def save_dual_value(self, value) -> None:
        dW = value[:, :-1]
        dz = value[:, -1]
        if self.axis == 0:
            dW = dW.T
            dz = dz.T
        if dW.shape[1] == 1:
            dW = np.squeeze(dW)
        self.dual_variables[0].save_value(dW)
        self.dual_variables[1].save_value(dz)

    def _dual_cone(self, *args):
        """Implements the dual cone of PowConeND See Pg 85
        of the MOSEK modelling cookbook for more information"""
        if args is None or args == ():
            scaled_duals = self.dual_variables[0]/self.alpha
            return PowConeND(scaled_duals, self.dual_variables[1],
                             self.alpha, axis=self.axis)
        else:
            return PowConeND(args[0]/self.alpha, args[1], self.alpha, axis=self.axis)
```

FIGURE 2.1: `save_dual_value` appropriately packs in the computed dual values into `W, z` while `_dual_cone` implements the `PowConeND`'s dual cone

After this, another major change that was needed was overriding the `save_dual_value::PowConeND` method — which is a method where the dual values that have been recovered inside of `invert` are actually stored onto the `dual_value` attribute on a `Constraint` object. Here, we needed to ensure that the recovered dual values were packed appropriately into the dual variables. The CVXPY API for the n-dimensional power cone is: `PowConeND(W, z, alpha, axis)` — note that this API allows for users to construct vectorized `PowConeND` constraints by passing in a matrix `W` and a vector `z` (with the `axis` argument dictating how these objects should be spliced).

We also implemented the `_dual_cone::PowConeND` method (newly introduced as part of my Google Summer of Code 2023 project) which houses a CVXPY implementation of the dual cone of the corresponding `Cone::Constrtaint` class.

With all of this in place, it turned out to be exceedingly simple to test this implementation since we could simply utilize the user-level KKT infrastructure that we had built during the summer i.e. we tested the KKT conditions against sample (feasible) problems involving `PowConeND`.

The core of the logic for the dual variable implementation, as described above, has been implemented inside of the `invert::Exotic2Common` and `save_dual_value::PowConeND` methods. They are reproduced in this chapter for completion's sake.

```
class Exotic2Common(Canonicalization):
    """
    other methods and fields
    """"""
     def invert(self, solution, inverse_data):
         pvars = {vid: solution.primal_vars[vid] for vid in inverse_data.id_map
                  if vid in solution.primal_vars}
         dvars = {orig_id: solution.dual_vars[vid]
                  for orig_id, vid in inverse_data.cons_id_map.items()
                  if vid in solution.dual_vars}

         if dvars == {}:
             #NOTE: pre-maturely trigger return of the method in case the problem
             # is infeasible (otherwise will run into some opaque errors)
             return Solution(solution.status, solution.opt_val, pvars, dvars,
                             solution.attr)

         dv = {}
         for cons_id, cons in inverse_data.id2cons.items():
             if isinstance(cons, PowConeND):
                 div_size = int(dvars[cons_id].shape[1] // cons.args[1].shape[0])
                 dv[cons_id] = []
                 for i in range(cons.args[1].shape[0]):
                     # Iterating over the vectorized constraints
                     dv[cons_id].append([])
                     tmp_duals = dvars[cons_id][:, i * div_size: (i + 1) * div_size]
                     for j, col_dvars in enumerate(tmp_duals.T):
                         if j == len(tmp_duals.T) - 1:
                             dv[cons_id][-1] += [col_dvars[0], col_dvars[1]]
                         else:
                             dv[cons_id][-1].append(col_dvars[0])
                     # dual value corresponding to `z`
                     dv[cons_id][-1].append(tmp_duals.T[0][-1])
                 dvars[cons_id] = np.array(dv[cons_id])

         return Solution(solution.status, solution.opt_val, pvars, dvars,
                         solution.attr)
```

FIGURE 2.2: Computing `PowConeND`'s duals from the dual values of it's constituent `PowCone3D` constraints

Chapter 3

The Operator Relative Entropy Cone and Semidefinite programming

3.1 Introduction

The *Operator Relative Entropy Cone* (henceforth referred to as the `OREC`) is a convex cone that features prominently in the modeling of several problems in quantum information.

In this chapter, we want to set the general theoretical backdrop so that the semidefinite approximation of the `OREC`, [11] given in the next chapter becomes a bit easier to digest. We will give some general theoretical background for the matrix logarithm, how it is used to generate the `OREC`, and give a brief overview of the rich field of semidefinite programming, which will be used to approximate the `OREC`.

Semidefinite programming problems are convex optimization problems that take on the following form, [22]:

$$\begin{aligned} \min_{X \in \mathcal{S}^n} \quad & \operatorname{tr}(C, X) \\ \text{subject to} \quad & \operatorname{tr}(A_i, X) = b_i \qquad \forall i \in [m] \\ & X \succeq 0 \end{aligned}$$

With it's dual being, [22]:

$$\begin{aligned} \max_{y \in \mathcal{R}^m} \quad & b^T y \\ \text{subject to} \quad & C = S + \sum_{i \in [m]} A_i y_i \\ & S \succeq 0 \end{aligned}$$

Specifically, we are interested in *semidefinite representations* (of 'size' d) of epigraphs convex functions f, i.e. cases when $\{(x,t) : f(x) \leq t\}$ can be expressed in the form of $\pi\left(L \cap \mathbb{H}_+^d\right)$ where π is a linear mapping and $\mathbb{H}_+^d$ is the set of all positive semidefinite, hermitian matrices. Understanding which convex sets and functions admit small semidefinite descriptions has been a significant area of research in recent history.

The caveat however, in relying on semidefinite descriptions, is that the feasible regions of semidefinite optimization problems are necessarily *semialgebraic sets* i.e. they can be expressed as a finite union of sets generated by polynomial inequalities. Which means that non-semialgebraic convex sets such as those generated by the logarithm (and it's matrix function generalizations) cannot be exactly modeled via the use of semidefinite programs.

The paper, [11], studies the problem of understanding which general convex sets and functions could be approximated with high accuracy with small semidefinite representations — that is to say, put explicitly, for some convex function f with a non-semialgebraic epigraph, how well can one construct a function that is not just convex, but also has a semidefinite representation of given size.

3.2 Univariate spectral functions

The route taken in [11] is to start from univariate functions and then generalize the same to the computation of matrix functions. If $g : \mathbb{R}_{++} \to \mathbb{R}$ then it's corresponding matrix function can be defined for the set of positive definite Hermitian matrices $\mathbb{H}_{++}^n$ by.

$$g(X) = U \operatorname{diag}(g(\lambda_1), \ldots, g(\lambda_n))U^*$$

Where, $X = U \operatorname{diag}(\lambda_1, \ldots, \lambda_n)U^*$ is an eigendecomposition of X.

Note that the matrix logarithm can be very naturally cast into the above template of spectral functions:

$$\begin{aligned} X &= U \operatorname{diag}(\lambda_1, \ldots, \lambda_n)U^* \\ \operatorname{logm}(X) &= U \operatorname{diag}(\log(\lambda_1), \ldots, \log(\lambda_n))U^* \end{aligned}$$

In addition to this, we also need a natural definition of convexity/concavity for spectral functions; this comes up as follows:

Definition 15. A function $g : \mathbb{R}_{++} \to \mathbb{R}$ is operator concave if the corresponding matrix function satisfies Jensen's inequality in the positive semidefinite order, i.e., to say:

$$g(\lambda X_1 + (1-\lambda)X_2) \succeq \lambda g(X_1) + (1-\lambda)g(X_2) \qquad \forall n \text{ and } X_1, X_2 \in \mathbb{H}^n_{++}$$

With the above in mind, we can now define the *matrix hypograph* of g as:
$\left\{(X,T) \in \mathbb{H}^n_{++} \times \mathbb{H}^n : g(X) \succeq T\right\}$

The logarithm is an operator concave function, and its operator concavity can be used to establish the joint convexity of the quantum relative entropy function, which features prominently in Chapter-6 and Chapter-7 (where we implement several applied problems from quantum information theory using this suite of `Atoms` and `Cones` that we have built)

3.3 Extensions to Bivariate matrix functions via Perspectives

Since we are primarily interested in bivariate functions of the like of the (Umegaki) quantum relative entropy, [11] then goes on to introduce a *noncommutative* notion of the perspective of a function.

Given a scalar-valued function $g : \mathbb{R}_{++} \to \mathbb{R}$ it's *perspective* transform is defined as $(x, y) \in \mathbb{R}^2_{++} \to yg(x/y)$.

The concavity of g also implies the concavity of its perspective transform. This notion can be extended to spectral functions (which are functions of hermitian PSD matrices).

Definition 16. The Non-Commutative perspective function corresponding to the function $g : \mathbb{R}_{++} \to \mathbb{R}$, where its translation to PSD, hermitian inputs is made by computing its corresponding spectral function, is a matrix function $P_g : \mathbb{H}^n_{++} \times \mathbb{H}^n_{++} \to \mathbb{H}^n$:

$$P_g(X,Y) = Y^{1/2} g\left(Y^{-1/2} X Y^{-1/2}\right) Y^{1/2}$$

The NCP is jointly concave in (X, Y) whenever g is operator concave, which is to say:

$$P_g(\lambda X_1 + (1-\lambda)X_2, \lambda Y_1 + (1-\lambda)Y_2) \succeq \lambda P_g(X_1, Y_1) + (1-\lambda)P_g(X_2, Y_2) \qquad \forall \lambda \in [0,1]$$

3.4 The `OREC`

All of the above mathematical machinery involving spectral functions of scalar-valued functions, followed by operator concavity thereof, their matrix hypographs, and finally capped off via the

introduction of the NCP to generalize the constructed matrix function to bivariate inputs leads us very elegantly towards the `OREC` as is captured in the definition below:

Definition 17. The `OREC` is the convex cone constructed from the matrix hypograph of the NCP of the matrix logarithm, which is computed via its spectral function form generated from the scalar logarithm, i.e., the `OREC` is the cone generated from the matrix hypograph of the *operator relative entropy* function, denoted by D_{op}:

$$D_{\text{op}}(X||Y) := -X^{1/2}\log(X^{-1/2}YX^{-1/2})X^{1/2}$$

The major throughline of [11] is that the approximations of the scalar logarithm function can be used to approximate the above, D_{op}, which in turn can be used to get semidefinite approximations of the all-important quantum relative entropy.

We discuss the actual nature of the quadrature-based approximations in Chapter-4

Chapter 4

An Approximate Canonicalization for the Operator Relative Entropy Cone

In Chapter3 we discussed the general background for how the OREC can be defined using all of these very tightly coupled pieces, each of which are essential in being able to write out a small semidefinite description.

In the following chapter, we wish to explain systematically, the methodology for arriving at the semidefinite description of the OREC is built upto in the paper, [11]

4.1 An integral representation of the scalar logarithm

The starting point for constructing these approximations is the observation that the (scalar) logarithm may be expressed as a definite integrand as follows:

$$\log(x) = \int_0^1 \frac{x-1}{t(x-1)+1} dt$$

We expect an integral representation of some generic concave function g, of the following form:

$$g(x) = \int_t f_t(x) d\nu(t)$$

Where ν is a positive measure and for any fixed t, $f_t(x)$ is a concave (rational) function of x that admits a semidefinite representation.

These kinds of integral representations are guaranteed to exist for certain kinds of operator concave functions (which the logarithm is, as was discussed in Chapter3), the following theorem formalizes this notion, [11].

Theorem 4 ([11], Theorem-4)**.** *If $g : \mathbb{R}_{++} \to \mathbb{R}$ is a non-constant operator monotone function then there is a unique probability measure μ supported on $[0, 1]$, such that:*

$$g(x) = g(1) + g'(1) \int_0^1 f_t(x) d\mu(t)$$

Where, f_t is the rational function mentioned in the above generic definition, which in the case of the logarithm happens to be: $f_t(x) = \frac{x-1}{t(x-1)+1}$

The above integrand representation can be naturally approximated via a quadrature rule with positive weights, which in turn gives us a generic way to approximate the concave function g:

$$g(x) \approx \sum_{j=1}^{m} w_j f_{t_j}(x)$$

In [11], the *Gaussian-Legendre* quadrature rule is used to estimate the weights (discussed in the next section)

Now, to cap this section off, we state the semidefinite representation of the epigraph of $x \to f_t(x)$:

$$f_t(x) \geq \tau \Leftrightarrow \begin{bmatrix} x - 1 - \tau & -\sqrt{t}\tau \\ -\sqrt{t}\tau & 1 - t\tau \end{bmatrix} \succeq 0$$

With the above in mind, we now define a function, r_m which will be our constructed quadrature approximation of the scalar logarithm, defined as:

$$r_m(x) := \sum_{j=1}^{m} w_j f_{t_j}(x) = \sum_{j=1}^{m} w_j \frac{x-1}{t_j(x-1)+1}$$

Here, $t_j \in [0, 1]$ are the quadrature nodes and $w_j > 0$ are the quadrature weights — m controls the number of nodes in the quadrature approximation (and will be an important parameter in our eventual implementation of the full `OREC` as well)

The key property of r_m is that it is concave *and* semidefinite-representable because of it being a nonnegative combination of functions that are each semidefinite representable, [11].

4.2 Gauss-Legendre Quadrature

A quadrature algorithm will output out a set of *nodes* and *weights* (corresponding to each node) over which the function will simply be evaluated and then summed over (with each evaluation being scaled by the appropriate weight). The Gauss-Legendre quadrature in specific, admits a

very rich and interesting formulation in terms of the *Legendre polynomials*, namely, the nodes of the G-L quadrature are nothing but the roots of the Legendre polynomials! Now comes the question of how does one generate these roots of the polynomials — as it turns out, there is a very special passage that exists, starting all the way from the *Arnoldi iteration* which gets specialised to the *Lanczos Algorithm* (by restricting the input matrix $\boldsymbol{A}$ to be *symmetric*) whose adaption to the continuous regime (i.e. *functions* $\leftrightarrow$ *vectors* and *operators* $\leftrightarrow$ *matrices*) yields us the required nodes, [20] is a great resource for reading more about the above.

Namely, we obtain the following recurrence (whose elements are to be subsequently used to construct the *Jacobi matrix* defined below), [20]:

$$\alpha_n = 0, \qquad \beta_n = \frac{1}{2}(1-(2n)^{-2})^{-1/2}.$$

Which in turn, leads us to the following definition.

Definition 18. The *Jacobi matrix* can be constructed from the above set of recurrence coefficients $\{\alpha_n\}$ and $\{\beta_n\}$ as:

$$T_n = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 & \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \cdots & 0 \\ 0 & \beta_2 & \alpha_3 & \ddots & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \beta_{n-1} \\ 0 & 0 & 0 & \cdots & \beta_{n-1} & \alpha_n \end{bmatrix}$$

With this, we have the following theorem, [20]:

Theorem 5 ([20], Theorem 37.4)**.** *With T_n being the $n \times n$ Jacobi matrix, let $T_n = VDV^T$ be an orthogonal diagonalization of V with $V = [v_1|v_2|\cdots|v_n]$ and $D = diag(\lambda_1, \cdots, \lambda_n)$. Then the nodes of the Gauss-Legendre quadrature formula are given by:*

$$x_j = \lambda_j, \qquad w_j = 2(v_j)_1^2, \qquad j = 1, \cdots, n$$

4.3 Improving the approximation via exponentiation

The logarithm satisfies the following standard functional equation:

$$\log(x^{1/2}) = \frac{1}{2}\log(x)$$

This means, that we can represent $\log(x)$ in terms of the logarithm of $\sqrt{x}$ — this is significant because the square-root operation brings points closer to $x = 1$ where the quadrature approximations are more accurate.

This additional exponentiation step also fits well with our existing mathematical machinery because the square root operation is: operator monotone, operator concave and semidefinite representable.

With this in mind, we now tweak the definition of our constructed approximate function r_m as follows:

$$r_{m,k}(x) = 2^k r_m(x^{1/2^k})$$

The approximation $r_{m,k}$ is to be understood as a composition of two steps:

1. Take the 2^kth-root of x and bring it closer to 1
2. Apply the approximation r_m and scale back by 2^k accordingly.

4.3.1 Error bounds for $r_{m,k}$

One can derive error bounds between $r_{m,k}$ and log. The correct way to study which, is by inspecting the Chebyshev coefficients of $t \to f_t(x)$ since r_m is defined in terms of Gaussian quadrature applied to a rational function $f_t(x)$

Proposition 1 ([11], Proposition-1)**.** *For any $x > 0$ we have:*

$$|r_{m,k}(x) - \log(x)| \leq 2^k \left|\sqrt{\kappa} - \sqrt{\kappa^{-1}}\right|^2 \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^{2m-1} \asymp 4.4^{-m(k+2)} \log(x)^{2m+1} \quad (k \to \infty)$$

By making appropriate choices of (m, k) in the above, we can obtain a result that tells us how the size of the semidefinite representation grows as the quality of the approximation improves

Theorem 6 ([11], Theorem-1)**.** *For any (fixed) $a > 1$ and any $\epsilon > 0$, there exists a function r such that $|r(x) - \log(x)| \leq \epsilon \quad \forall x \in [1/a, a]$ and r has a semidefinite representation of size $\mathcal{O}\left(\sqrt{\log(1/\epsilon)}\right)$.*

4.3.2 Generalization to matrix functions

All of the above machinery holds just as well for matrix functions as well, this is made possible because:

f_t, the rational function in terms of which we express the quadrature approximation, r_m is operator concave and the matrix hypograph generated by its spectral function equivalent admits a *small* semidefinite description

An important result to note here is the semidefinite representation of P_{f_t}, where $P(\cdot)$ is the NCP of a univariate matrix function defined in Chapter3, we reproduce it below for completion's sake:

Proposition 2 ([11], Proposition-8)**.** *If $t \in [0,1]$ then the perspective P_{f_t} of f_t is jointly matrix concave since:*

$$P_{f_t}(X,Y) \succeq T \quad \textit{and} \quad X,Y \succ 0 \Leftrightarrow \begin{bmatrix} X-Y & 0 \\ 0 & Y \end{bmatrix} - \begin{bmatrix} T & \sqrt{t}T \\ \sqrt{t}T & tT \end{bmatrix} \succeq 0 \textit{ and } X,Y \succ 0$$

Proposition 3 ([11], Proposition-2)**.** *For $t \in [0,1]$, let $f_t = \frac{x-1}{t(x-1)+1}$. Then (the spectral function equivalent of) f_t is operator concave. Its matrix hypograph admits the following semidefinite description:*

$$f_t(X) \succeq T \quad \textit{and} \quad X \succ 0 \Leftrightarrow \begin{bmatrix} X-I & 0 \\ 0 & I \end{bmatrix} - \begin{bmatrix} T & \sqrt{t}T \\ \sqrt{t}T & tT \end{bmatrix} \succeq 0 \quad \textit{and} \quad X \succ 0$$

From the above result, one can deduce the following:

Proposition 4 ([11], Proposition-3)**.** *The (spectral function equivalent of) $r_{m,k}$ is operator concave*

With this, we are ready to present the full semidefinite representation of the `OREC`

4.4 A semidefinite representation for the `OREC`

The `OREC` is the epigraph associated with the operator relative entropy matrix function (which is the NCP of the negative logarithm).

$$D_{\text{op}}(X||Y) := -X^{1/2}\log(X^{-1/2}YX^{-1/2})X^{1/2}$$
$$K^n_{\text{re}} = \text{cl}\left\{(X,Y,T) \in \mathbb{H}^n_{++} \times \mathbb{H}^n_{++} \times \mathbb{H}^n : T \succeq D_{\text{op}}(X||Y)\right\}$$

Hence, the key to approximating the `OREC` is to approximate the D_{op} — which, according to the discussion so far, can very naturally be done by taking the NCP of $-r_{m,k}$, i.e. to say our approximation of the `OREC` is:

$$K^n_{m,k} = \left\{(X,Y,T) \in \mathbb{H}^n_{+} \times \mathbb{H}^n_{+} \times \mathbb{H}^n : T \succeq -P_{r_{m,k}}\right\}$$

Where P is the NCP of a univariate matrix function

The main theorem of [11], which gives the semidefinite representation of $r_{m,k}$ is reproduced in full below. Before this, we need to define the notion of a *weighted matrix geometric mean.*

Definition 19. For any $0 < h < 1$ the h-weighted geometric mean of $A, B \succ 0$ is denoted $A\#_h B$ and defined by:

$$A\#_h B := A^{1/2}\left(A^{-1/2}BA^{-1/2}\right)^h A^{1/2}$$

Note that $A\#_h B$ is the NCP of the power function $x \to x^h$. All of the *usual* properties defined so far in this work also hold for the WMGM (operator concavity in (A, B) and semidefinite representability)

Theorem 7 ([11], Theorem-3)**.** *A triple of matrices (X, Y, T) belongs to $K^n_{m,k}$ if and only if*

$$\begin{cases} Z_0 = Y, & \begin{bmatrix} Z_i & Z_{i+1} \\ Z_{i+1} & X \end{bmatrix} \succeq 0 \quad (i = 0, \ldots, k-1) \\ \sum_{j=1}^{m} w_j T_j = -2^{-k} T, & \begin{bmatrix} Z_k - X - T_j & -\sqrt{t_j}T_j \\ -\sqrt{t_j}T_j & X - t_j T_j \end{bmatrix} \succeq 0 \quad (j = 1, \ldots m) \end{cases}$$

holds for some $T_1, \ldots, T_m, Z_0, \ldots, Z_k \in \mathbb{H}^n$.

We sketch the larger structure of the proof of this *very* impressive result below.

Proof. We are interested in being able to represent $P_{r_{m,k}}$ in a form where we can easily derive its semidefinite description, this is helped by making the following observation:

$$P_{r_{m,k}}(Y, X) = 2^k P_{r_m}((X\#_{2^{-k}}Y), X)$$

The above-stated semidefinite representation follows from the following three facts:

1. *Semidefinite representation of the WMGM*: For any $X, Y \succ 0$ and $V \in \mathbf{H}^n$ and $k \geq 1$ we have $X\#_{2^{-k}}Y \succeq V$ if an only if there exists $Z_0, \ldots, Z_k \in \mathbf{H}^n$ that satisfy:

$$Z_0 = Y, Z_k = V \quad \text{and} \quad \begin{bmatrix} Z_i & Z_{i+1} \\ Z_{i+1} & X \end{bmatrix} \succeq 0 \quad (i = 0, \ldots k-1)$$

   This representation is restricted to the case of having $h = 1/2^k$. This construction hinges on the fact that $X\#_{2^{-k}}Y$ can be expressed in terms of k nested geometric means as $X\#_{1/2}(X\#_{1/2}(\ldots(X\#_{1/2}Y)))$

2. *Semidefinite representation of* P_{r_m}: For any $V, X \succ 0$ and $T \in \mathbf{H}^n$ we have $P_{r_m}(V, X) \succeq T$ if and only if there exists $T_1, \ldots, T_m$ that satisfy:

$$\sum_{j=1}^{m} w_j T_j = T \quad \text{and} \quad \begin{bmatrix} V - X - T_j & -\sqrt{t_j} T_j \\ -\sqrt{t_j} T_j & X - t_j T_j \end{bmatrix} \succeq 0 \quad (j = 1 \ldots, m)$$

   The above representation follows from the semidefinite representation of P_{f_t} (stated above) and $\sum_{j=1}^{m} w_j f_{t_j}$

3. P_{r_m} is monotone in its first argument. This is also easily established from the monotonicity of r_m

Combining the above three facts into the WMGM expression for the $P_{r_{m,k}}$ yields the stated semidefinite representation in the theorem. □

Chapter 5

Implementing the OREC within CVXPY

This chapter details the implementation of the two constraint classes that were added to CVXPY as a part of this undertaking, namely, `RelEntrConeQuad` and `OpRelEntrConeQuad`. In terms of Chapter-4's terminology, these are $K^{1}_{\mathrm{m,\,k}}$ and $K^{n}_{\mathrm{m,\,k}}$ respectively.

The reason we decided to start from the scalar relative entropy was two-fold: (1) As an initial exercise towards familiarizing ourselves with the structure of the semidefinite descriptions in a far simpler environment (only real inputs are allowed), (2) Be able to trivially check the correctness of our implementation because of the equivalence of the `sREC` and the exponential cone (which made a brief appearance back in Chapter-1). Additionally, both of the approximations share the Gauss-Legendre quadrature implementation.

Note that both of these are `Constraint` classes (they are also members of the `Cone::Constraint` class), which is CVXPY's abstraction for a constraint in an optimization problem.

One of the principal difficulties in implementing this entire quantum information suite within CVXPY was that of *testing*. Compared to the abundance of literature and example problems available on classical cones like the exponential cone, the second-order cone, etc., there is a marked lack of the same for the `OREC`. While we opted to leverage the correctness of the original `CVXQUAD` (Matlab) implementation for verifying the correctness of solutions later on in the development of these tools, we had to rely on crafting clever problems involving our to-be-tested objects whose correctness could otherwise be verified when we had just begun.

5.1 The `sREC`

The `sREC`, or, K_{re}^1 is just a particular instantiation of the `OREC` which was discussed in Chapter-3:

$$K_{\text{re}} := \text{cl}\left\{(x, y, \tau) \in \mathbb{R}_{++} \times \mathbb{R}_{++} \times \mathbb{R} : x \log(x/y) \leq \tau\right\}$$

Which we are approximating via $K_{m,k}$ (only the scalar variant of every function makes an appearance in all of these quantities):

$$K_{m,k} := \left\{(x, y, \tau) \in \mathbb{R}_{++}^2 \times \mathbb{R} : x r_{m,k}(x/y) \leq \tau\right\}$$

The relationship between the `sREC` and the exponential cone is very easy to observe from their definitions, namely, we only require a permutation and a sign change of the arguments to transform to the other. Specifically: If $(x_1, x_2, x_3) \in K_{\text{exp}} \implies (x_2, x_3, -x_1) \in K_{\text{re}}$. This permutation has been implemented in the `as_expconequad(m, k)` method under the `ExpCone` class and is the method that provides the primary interface for working with this quadrature approximation variant of the exponential cone. One can refer to [12] for an interesting study of the `sREC` and the exponential cone in the context of the root-finding problem.

Now, an interesting observation that can be made in the case of the `sREC` is the fact that 2×2 SDP constraints can be very easily reformulated as SOCs, specifically, in terms of a rotated-second-order-cone because:

$$\begin{bmatrix} x & y \\ y & z \end{bmatrix} \succeq 0 \Leftrightarrow x \geq 0, z \geq 0, xz \geq y^2$$

The last of which is a rotated SOC constraint. To this end, there exists the `quad_over_lin(X, y)` atom in CVXPY which computes the quantity $\sum_i \frac{X_{ij}^2}{y}$ whose epigraph i.e. `quad_over_lin(X, y)` $\leq t$ is the required quantity. The issue however being that the atom has not been vectorized properly for vector inputs $\mathbf{y}$.

We explore how we get around this in one of the sections next — before that, we make a brief detour and discuss the implementation of the Gauss-Legendre quadrature, the relevant theoretical background for which was provided in Chapter-4.

5.1.1 Implementing the Gauss-Legendre quadrature

As you may recall from Chapter-4, the Gauss-Legendre quadrature requires the construction Jacobi matrix, from whose eigendecomposition we construct the nodes and weights of the quadrature algorithm.

The construction of the Jacobi Matrix requires the computation of the betas and their appropriate arrangement as part of a matrix.

All of this structure can be very elegantly captured within vanilla numpy as shown below, [19]:

```
def gauss_legendre(n):
    """
    Helper function for returning the weights and nodes for an
    n-point Gauss-Legendre quadrature on [0, 1]
    """
    beta = 0.5/np.sqrt(np.ones(n-1)-(2*np.arange(1, n, dtype=float))**(-2))
    T = np.diag(beta, 1) + np.diag(beta, -1)
    D, V = np.linalg.eigh(T)
    x = D
    x, i = np.sort(x), np.argsort(x)
    w = 2 * (np.array([V[0][k] for k in i]))**2
    x = (x + 1)/2
    w = w/2
    return w, x
```

FIGURE 5.1: Gauss-Legendre quadrature, python implementation

5.1.2 Vectorizing `quad_over_lin`$(\cdot, \cdot)$

Vectorizing the input over `y` turned out to be somewhat of an involved process. We opted to move away from using `quad_over_lin` out of the box, and chose to write our own function for constraining a set of variables to the rotated SOC since CVXPY's `SOC::Cone::Constraint` class implements a vanilla SOC constraint.

The implementation for the same hinges on the following two key ideas:

1. If $(y, t, 2x) \in rSOC \Leftrightarrow (y + t, y - t, 2x) \in SOC$
2. The way the SOC constraint is vectorized in CVXPy is as: Assumes t is a vector the same length as X's columns (rows) for `axis == 0` (1).

```
def rotated_quad_cone(X: cp.Expression, y: cp.Expression, z: cp.Expression):
    """
    For each i, enforce a constraint that
        (X[i, :], y[i], z[i])
    belongs to the rotated quadratic cone
        { (x, y, z) : || x ||^2 <= y z, 0 <= (y, z) }
    This implementation doesn't enforce (x, y) >= 0!
    That should be imposed by the calling function.
    """
    m = y.size
    assert z.size == m
    assert X.shape[0] == m
    if len(X.shape) < 2:
        X = cp.reshape(X, (m, 1))
    ######################################
    #   quad_over_lin := sum_{i} x^2_{i} / y
    #   t = Variable(1,) is the epigraph variable.
    #   Becomes a constraint
    #   SOC(t=y + t, X=[y - t, 2*x])
    ######################################
    soc_X_col0 = cp.reshape(y - z, (m, 1))
    soc_X = cp.hstack((soc_X_col0, 2*X))
    soc_t = y + z
    con = cp.SOC(t=soc_t, X=soc_X, axis=1)
    return con
```

FIGURE 5.2: Code for constraining $(x, y, z) \in rSOC$

5.2 The `OREC`

Recall the definitions of the operator relative entropy and it's epigraph, the `OREC` from sec4.4.

$$D_{\text{op}}(X||Y) := -X^{1/2}\log(X^{-1/2}YX^{-1/2})X^{1/2}$$
$$K_{\text{re}}^n = \text{cl}\left\{(X, Y, T) \in \mathbb{H}_{++}^n \times \mathbb{H}_{++}^n \times \mathbb{H}^n : T \succeq D_{\text{op}}(X||Y)\right\}$$

The implementation of the canonicalization method for the `OREC` was tedious from a book-keeping perspective, but ultimately because of CVXPY's elegant abstractions, more-or-less follows one-to-one from the semidefinite description of the $K_{m,k}^n$ provided in Chapter-4, especially with the `gauss_legendre` routine in place.

The additional step that had to be done with `OpRelEntrConeQuad`'s implementation was to tell CVXPY what happens when the inputs to the cone are hermitian in nature (and not just real and symmetric).

For some background, CVXPY has an abstraction of `Reduction` classes which *reduce* various aspects of a problem down into a more fundamental representation (one that lower level solvers,

like `CLARABEL` or `MOSEK` can accept). The `Reduction` class that tells CVXPY what to do when the inputs to a particular function or a constraint are complex in nature, is the appropriately named `Complex2Real::Reduction` class.

So, to make sure CVXPY knew what to do when `OpRelEntrConeQuad` received complex, hermitian input, we had to implement a separate canonicalization method that converted the complex, hermitian inputs received by the `OpRelEntrConeQuad` constructor, to real, symmetric inputs that could be processed by the actual canonicalization routine. This is done by expanding every argument (say A) as $A \to B = \begin{bmatrix} \mathrm{Re}(A) & -\mathrm{Im}(A) \\ \mathrm{Im}(A) & \mathrm{Re}(A) \end{bmatrix}$. The core implementation of `OREC`'s canonicalization method is reproduced below:

```
def OpRelEntrConeQuad_canon(con: OpRelEntrConeQuad, args)
                            -> Tuple[Constraint, List[Constraint]]:
    k, m = con.k, con.m
    X, Y = con.X, con.Y
    assert X.is_real()
    assert Y.is_real()
    assert con.Z.is_real()
    Zs = {i: Variable(shape=X.shape, symmetric=True) for i in range(k+1)}
    Ts = {i: Variable(shape=X.shape, symmetric=True) for i in range(m+1)}
    constrs = [Zero(Zs[0] - Y)]
    if not X.is_symmetric():
        ut = upper_tri(X)
        lt = upper_tri(X.T)
        constrs.append(ut == lt)
    if not Y.is_symmetric():
        ut = upper_tri(Y)
        lt = upper_tri(Y.T)
        constrs.append(ut == lt)
    if not con.Z.is_symmetric():
        ut = upper_tri(con.Z)
        lt = upper_tri(con.Z.T)
        constrs.append(ut == lt)
    w, t = gauss_legendre(m)
    lead_con = Zero(cp.sum([w[i] * Ts[i] for i in range(m)]) + con.Z/2**k)

    for i in range(k):
        #     [Z[i]  , Z[i+1]]
        #     [Z[i+1], x     ]
        constrs.append(cp.bmat([[Zs[i], Zs[i+1]], [Zs[i+1].T, X]]) >> 0)

    for i in range(m):
        off_diag = -(t[i]**0.5) * Ts[i]
        # The following matrix needs to be PSD.
        #     [ Z[k] - x - T[i] , off_diag      ]
        #     [ off_diag        , x - t[i]*T[i] ]
        constrs.append(cp.bmat([[Zs[k] - X - Ts[i], off_diag],
                                [off_diag.T, X-t[i]*Ts[i]]]) >> 0)

    return lead_con, constrs
```

FIGURE 5.3: Canonicalization routine for the OREC

5.2.1 Testing the implementation

As I alluded to earlier, one of the principal challenges in the implementation of the `OREC` was being able to write verifiable tests for the same. For the same, we resorted to the clever construction of programs whose correctness could otherwise be verified (i.e. a reference solution could pre-maturely be computed either directly, or via a reformulation in terms of existing CVXPY functionality).

We construct such verifiable tests based off of a clever choice of the nature of the objective function with the only constraint in the problem being a `OpRelEntrConeQuad` constraint (with some feasibility constraints placed on the eigenvalues of (A, B)). Namely, we make the following observation:

Lemma 2. *If $(A, B, T) \in K_{re}^n$, i.e. we have $T \succeq D_{op}(A, B)$. We have $T = D_{op}(A, B)$ for any objective that is an increasing function of the eigenvalues.*

Proof. This follows naturally from the characterization of the PSD cone in terms of every constituent matrix having positive eigenvalues, namely:

$$\mathcal{S}_{++}^n = \{X \in \mathcal{S}^n | \lambda_i(X) \geq 0, i \in [n]\}$$

If the sole significant constraint in our problem is: $T \succeq D_{\text{op}}$, we have $(T - D_{\text{op}}) \succeq 0 \implies \lambda_i(T - D_{\text{op}}) \geq 0$. Assuming the objective is to minimize $\text{tr}(T)$ which is the sum of all of the eigenvalues of T, it is clear that such a problem has a trivial solution at $T = D_{\text{op}}$. □

Also note that we can compute the objective easily in such a case. Say we have the tr(.) as the objective function, then we can write: $\text{tr}(T) = \text{tr}(T - D_{\text{op}}) + \text{tr}(D_{\text{op}})$, where $\text{tr}(T - D_{\text{op}}) \geq 0$ because $T \succeq D_{\text{op}}$, and since at optimality we have $T = D_{\text{op}}$, we have $\text{tr}(T) = \text{tr}(D_{\text{op}})$

With the above structure in mind, we decided to construct two different kinds of test cases based on the way we computed D_{op}:

1. (A, B) *commute.* In this case, (A, B) are constructed to share eigenvectors but have different eigenvalues to ensure that they commute during the computation of the D_{op} (recall that the D_{op} is the NCP of the matrix logarithm). In such a case, the D_{op} admits a particularly simple form, namely: $D_{op}(A, B) = U \,\text{diag}(a \log(a/b))U^{-1}$ (where a, b are the set of eigenvalues of (A, B) respectively)
2. (A, B) *do NOT commute.* In this case, (A, B) are constructed independently. The $T = D_{\text{op}}$ condition still holds, but in this case, instead of being able to compute the reference solutions in place, we leverage the CVXQUAD (matlab) implementation for the same.

Chapter 6

Implementing important `Atoms` using the `OREC`

In this chapter, we discuss arguably what the entirety of this project led up to, and that is the implementation of the all-important Von-Neumann Entropy (`von_neumann_entr`) and Quantum Relative Entropy (`quantum_rel_entr`) as CVXPY `Atoms` i.e. as functions which can be used as part of constructing the objective when writing out optimization problems in CVXPY.

Definition 20. The Von-Neumann Entropy of a quantum state ρ is defined as, [18]:

$$S(\rho) := -\operatorname{tr}(\rho \log(\rho))$$

Definition 21. The Quantum Relative entropy of two quantum states, ρ, σ is defined as, [18]:

$$D(\rho||\sigma) := \operatorname{tr}[\rho(\log \rho - \log \sigma)]$$

6.1 Implementing the `VNE`

The `VNE` can be implemented routinely with access to a working `OpRelEntrConeQuad` implementation, but as it so happens, the `VNE` also admits an *exact* canonicalization which is described in the paper [6].

As of today, the API for `von_neumann_entr` has been designed to ensure users can very naturally use either of the two possible canonicalization pathways. We default to using the exact pathway and use the approximate if the user passed in the (m, k) tuple when instantiating `von_neumann_entr::Atom`.

6.1.1 The *exact* canonicalization

We reproduce *Proposition-4* from [6] here:

Theorem 8 ([6], Proposition-4)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex function that is invariant under permutation of its argument, and let $g : \mathbb{S}^n \to \mathbb{R}$ be the convex function defined as $g(\boldsymbol{N}) = f(\lambda(\boldsymbol{N}))$. Here $\lambda(\boldsymbol{N})$ refers to the list of eigenvalues of the matrix $\boldsymbol{N}$. Then we have that:*

$$
\begin{aligned}
& g(\boldsymbol{N}) \leq t \\
& \Updownarrow \\
& f(\boldsymbol{x}) \leq t \\
& \boldsymbol{x}_1 \geq \cdots \geq \boldsymbol{x}_n \\
\exists \boldsymbol{x} \in \mathbb{R}^n s.t. \quad & s_r(\boldsymbol{N}) \geq \boldsymbol{x}_1 + \cdots + \boldsymbol{x}_r, r = 1, \ldots, n-1 \\
& Tr(\boldsymbol{N}) = \boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n
\end{aligned}
$$

where, $s_r(\boldsymbol{N})$ is the sum of the r largest eigenvalues of $\boldsymbol{N}$

6.1.2 An `OREC` description of the `VNE`

The epigraph of `von_neumann_entr` can be expressed in terms of the `OREC`, the basis for which is the observation that:

$$S = \mathrm{tr}(D_{op}(X||I)) \tag{6.1}$$

From the above representation, a very natural representation of the `von_neumann_entr` in terms of `OpRelConeQuad` follows, i.e.:

$$(N, T) \in \{(X, Q) : Q \succeq S\} \iff (N, I, T) \in K_{re}^n$$

6.2 Implementing the `QRE`

The `QRE` is perhaps the most important `Atom` that can be implemented for applications in quantum information theory when one has a working `OREC` implementation.

The following proposition shows how the epigraph of the `QRE` can be expressed in terms of the `OREC`, [11]:

```
def von_neumann_entr_canon(expr, args):
    N = args[0]
    assert N.is_real()
    n = N.shape[0]
    x = Variable(shape=(n,))
    t = Variable()

    # START code that applies to all spectral functions #
    constrs = []
    for r in range(1, n):
        # lambda_sum_largest(N, r) <= sum(x[:r])
        expr_r = lambda_sum_largest(N, r)
        epi, cons = lambda_sum_largest_canon(expr_r, expr_r.args)
        constrs.extend(cons)
        con = NonNeg(sum(x[:r]) - epi)
        constrs.append(con)

    # trace(N) \leq sum(x)
    con = trace(N) == sum(x)
    constrs.append(con)

    # trace(N) == sum(x)
    con = Zero(trace(N) - sum(x))
    constrs.append(con)

    # x[:(n-1)] >= x[1:]
    #   x[0] >= x[1],  x[1] >= x[2], ...
    con = NonNeg(x[:(n - 1)] - x[1:])
    constrs.append(con)

    # END code that applies to all spectral functions #

    # sum(entr(x)) >= t
    hypos, entr_cons = entr_canon(x, [x])
    constrs.extend(entr_cons)
    con = NonNeg(sum(hypos) - t)
    constrs.append(con)

    return t, constrs
```

FIGURE 6.1: CVXPY implementation of the *exact* representation of the `VNE`

```
def quantum_rel_entr_canon(expr, args):
X, Y = args
n = X.shape[0]
Imat = np.eye(n)
e = Imat.ravel().reshape(n ** 2, 1)
assert X.is_symmetric()
assert Y.is_symmetric()
first_arg = cp.atoms.affine.wraps.symmetric_wrap(kron(X, Imat))
second_arg = cp.atoms.affine.wraps.symmetric_wrap(kron(Imat, Y))
epi = Variable(shape=first_arg.shape, symmetric=True)
orec_con = OpRelEntrConeQuad(first_arg, second_arg, epi,
                             expr.quad_approx[0], expr.quad_approx[1]
)
main_con, aux_cons = cp.reductions.cone2cone.approximations.OpRelEntrConeQuad_canon(
    orec_con, None
)
constrs = [main_con] + aux_cons
return  e.T @ epi @ e, constrs
```

FIGURE 6.2: `OREC` based canonicalization method for the `quantum_rel_entr`

Proposition 5. *Let D be the relative entropy and D_{op} be the operator relative entropy. Then for any $A, B \succ 0$ the following identity holds:*

$$D(A||B) = \phi\left(D_{op}\left(A \otimes I || I \otimes \overline{B}\right)\right)$$

Where ϕ is the unique linear map from $\mathbb{C}^{n^2 \times n^2}$ to $\mathbb{C}$ that satisfies $\phi(X \otimes Y) = tr\left[XY^T\right]$, and $\overline{B}$ is the entrywise complex conjugate of B

The above identity relating the `QRE` and the operator relative entropy function allow us to express the `QRE`'s epigraph in terms of the `OREC` as follows:

Corollary 1. *For any $A, B \succ 0$ and $\tau \in \mathbb{R}$ we have:*

$$D(A||B) \leq \tau \iff \exists T \in \mathbf{H}^{n^2} : \left(A \otimes I, I \otimes \overline{B}, T\right) \in K_{re}^{n^2} \text{ and } \phi(T) \leq \tau$$

An important point to note with the implementation of the above is the exact nature of the ϕ map. The linear map in ϕ is simply given by: $\phi(Z) = w^* Z w$ for $Z \in \mathbb{C}^{n^2 \times n^2}$, where $w \in \mathbb{C}^{n^2}$ is the vector obtained by stacking the columns of the $n \times n$ identity matrix. It follows taht ϕ is a positive linear map, in the sense that if $Z \succeq 0$ then $\phi(Z) \geq 0$

Chapter 7

Quantum Information problems in CVXPY

In this chapter, we do a brief study of some applied problems that can be modeled using this quantum information suite in CVXPY.

Remark: The structure and prose of this chapter borrows *heavily* from the paper [10], with the addition of CVXPY/python-specific implementation details.

7.1 Capacity of a Classical to quantum channel

For a finite input alphabet χ and a finite-dimensional Hilbert space A, a classical-quantum channel is a mapping $\Phi : \chi \to D(A)$ which maps symbols $x \in \chi$ to density operators $\Phi(x)$. The capacity of such a channel. The capacity of such a channel can be computed as the solution of an optimization problem:

$$\begin{aligned} \underset{p \in \mathbb{R}^{\chi}}{\text{maximize}} \quad & H\left(\sum_{x \in \chi} p(x)\Phi(x)\right) - \sum_{x \in \chi} p(x) H\left(\Phi(x)\right) \\ \text{subject to} \quad & p \geq 0, \sum_{x \in \chi} p(x) = 1 \end{aligned}$$

Where $H(.)$ is the Von Neumann entropy.

The above problem can be implemented fairly straightforwardly in CVXPY using a mixture of standard CVXPY functionality and the `von_neumann_entr` atom. It also requires the `randRho`

routine from the qubit package which generates random density matrices (which in turn relies on the `randH` routine which generates random hermitian matrices)

```
def randH(n: int):
    A = np.random.randn(n, n) + 1j * np.random.randn(n, n)
    return (A + A.conj().T)/2

def randRho(n: int):
    p = 10 * randH(n)
    p = (p @ p.conj().T)/np.trace(p @ p.conj().T)
    return p

import numpy as np
import cvxpy as cp

rho1 = randRho(2)
rho2 = randRho(2)
H1 = cp.von_neumann_entr(rho1)
H2 = cp.von_neumann_entr(rho2)

p1 = cp.Variable()
p2 = cp.Variable()

obj = cp.Maximize((cp.von_neumann_entr(p1 * rho1 + p2 * rho2)
                   - p1 * H1 - p2 * H2)/np.log(2))
cons = [
    p1 >= 0,
    p2 >= 0,
    p1 + p2 == 1
]

prob = cp.Problem(obj, cons)
prob.solve(solver='MOSEK')
```

FIGURE 7.1: Capacity of a cq-channel

7.2 Entanglement-assisted classical capacity

The *entanglement-assisted classical capacity* of a quantum channel Φ quantifies the amount of classical bits that can be transmitted reliable through it, if, the receiver and the transmitter are allowed to share an arbitrary entangled state.

We require the notion of the *mutual-information* of the channel Φ for some input state ρ:

Definition 22. Let $U : A \to B \otimes E$ be a Stinespring isometry for Φ with environment E i.e. such that $\Phi(X) = \mathrm{tr}_E [UXU^*]$ for any operator X on A. Then $I(\rho, \Phi)$ is defined as:

$$I(\rho, \Phi) := H(B|E)_{U\rho U^*} + H(B)_{U\rho U^*}$$

Where $H(B|E)$ denotes the conditional entropy.

Proposition 6. *The mutual information, $I(\rho, \Phi)$ defined above is concave in ρ*

Coming back to the computation of the entanglement-assisted classical capacity. The problem can be shown to admit the following maximization expression:

$$C_{\mathrm{ea}} = \max_{\rho \in D(A)} I(\rho, \Phi)$$

The above formula is the quantum analogue of the formula for the Shannon capacity of a classical channel.

This can be implemented within CVXPY using a combination of traditional CVXPY functionalities and most notably, the `quantum_cond_entr`, `von_neumann_entr` and `partial_trace` atoms.

The quantum conditional entropy admits a definition in terms of the `QRE` and the partial trace operator. Its CVXPY implementation is given below:

```
def quantum_cond_entr(rho, dim: list[int], sys=0):
    if sys == 0:
        composite_arg = kron(np.eye(dim[0]),
                             partial_trace(rho, dim, sys))
        return -quantum_rel_entr(rho, composite_arg)
    elif sys == 1:
        composite_arg = kron(partial_trace(rho, dim, sys),
                             np.eye(dim[1]))
        return -quantum_rel_entr(rho, composite_arg)
```

FIGURE 7.2: `quantum_cond_entr` in CVXPY

7.3 Quantum capacity of degradable channels

There are a few definitions required to set this problem up, we reproduce them from [10].

Definition 23. If Φ is a quantum channel from A to B with environment E and an isometry representation $U : A \to B \otimes E$, then the complimentary channel, $\Phi^c : L(A) \to L(E)$ is given by:

$$\Phi^c(\rho) = \operatorname*{tr}_B [U\rho U^*]$$

```
import cvxpy as cp
import numpy as np

na, nb, ne = (2, 2, 2)
AD = lambda gamma: np.array([[1, 0], [0, np.sqrt(gamma)], [0, np.sqrt(1-gamma)], [0, 0]])
U = AD(0.2)

rho = cp.Variable(shape=(na, na), hermitian=True)
obj = cp.Maximize((cp.quantum_cond_entr(U @ rho @ U.conj().T, [nb, ne]) +
                   cp.von_neumann_entr(cp.partial_trace(U @ rho @ U.conj().T,
                   [nb, ne], 1)))/np.log(2))
cons = [
    rho >> 0,
    cp.trace(rho) == 1
]
prob = cp.Problem(obj, cons)
prob.solve()
```

FIGURE 7.3: Entanglement-assisted classical capacity of a quantum channel

Definition 24. The coherent information of a channel Φ for the input ρ is defined as:

$$I_c(\rho, \Phi) := H\left(\Phi(\rho)\right) - H\left(\Phi^c(\rho)\right)$$

The unassisted quantum capacity $Q(\Phi)$ of quantum channels Φ is the number of qubits that can be reliably transmitted over Φ — it can be computed via the following expression:

$$Q(\Phi) = \lim_{n\to\infty} \max_{\rho^{(n)}} \frac{1}{n} I_c(\rho^{(n)}, \Phi^{\otimes n})$$

Writing this problem out in CVXPY required the implementation of the `applychan` method from the [7] package (whose relevant chunk is shown below). Additionally, this required the `quantum_cond_entr` (which depends on the `QRE` as discussed above).

7.4 Relative Entropy of entanglement

The *relative entropy of entanglement* is defined as the distance from a bipartite state on $D(A\otimes B)$, ρ, to the set of all separable states on $A \otimes B$ (`Sep`):

$$\texttt{REE}(\rho) = \min_{\tau\in\texttt{Sep}} D(\rho||\tau)$$

```
def applychan(chan: np.array, rho: cp.Variable, rep: str, dim: tuple[int, int]):
    tol = 1e-10

    dimA, dimB, dimE = None, None, None
    match rep:
        case 'choi2':
            dimA, dimB = dim
        case 'isom':
            dimA = chan.shape[1]
            dimB = dim[1]
            dimE = int(chan.shape[0]/dimB)
            pass

    match rep:
        case 'choi2':
            arg = chan @ kron(rho.T, np.eye(dimB))
            rho_out = partial_trace(arg, [dimA, dimB], 0)
            return rho_out
        case 'isom':
            rho_out = partial_trace(chan @ rho @ chan.conj().T, [dimB, dimE], 1)
            return rho_out

na, nb, ne, nf = (2, 2, 2, 2)
AD = lambda gamma: np.array([[1, 0],[0, np.sqrt(gamma)],[0, np.sqrt(1-gamma)],[0, 0]])
gamma = 0.2
U = AD(gamma)

W = AD((1-2*gamma)/(1-gamma))

Ic = lambda rho: cp.quantum_cond_entr(
    W @ applychan(U, rho, 'isom', (na, nb)) @ W.conj().T,
    [ne, nf], 1
)/np.log(2)

rho = cp.Variable(shape=(na, na), hermitian=True)
obj = cp.Maximize(Ic(rho))
cons = [
    rho >> 0,
    cp.trace(rho) == 1
]
prob = cp.Problem(obj, cons)
prob.solve()
```

FIGURE 7.4: Quantum capacity of degradable channels

The set of all separable states is infamously hard to characterize, a popular relaxation for it is to replace $\rho \in \texttt{Sep}$ with imposing τ to have a positive partial transpose (the *PPT relaxation* of this problem as it's called)

$$\texttt{REE}^{(1)}(\rho) = \min_{\tau \in \text{PPT}} D(\rho||\tau)$$

The CVXPY implementation for the same follows below:

```
na, nb = (2, 2)
rho = randRho(na * nb)
tau = cp.Variable(shape=(na * nb, na * nb), hermitian=True)
obj = cp.Minimize(cp.quantum_rel_entr(rho, tau, (3,3))/np.log(2))
cons = [tau >> 0, cp.trace(tau) == 1, cp.partial_transpose(tau, [na, nb], 1) >> 0]
prob = cp.Problem(obj, cons)

prob.solve()
```

FIGURE 7.5: Relative entropy of entanglement

Bibliography

[1] MOSEK ApS. *Dual variables for the Power Cone with Long LHS*. https://groups.google.com/g/mosek/c/fmQFe0C4Ups?pli=1.

[2] MOSEK ApS. *MOSEK Modeling Cookbook, v3.3.0*. 2023.

[3] Amir Beck. *First-Order Methods in Optimization*. SIAM, 2017.

[4] Aharon Ben-Tal and Arkadi Nemirovski. *Lectures on Modern Convex Optimization*. 2022.

[5] Stephen Boyd. *EE364a: Convex Optimization I, Lecture Slides*. https://web.stanford.edu/class/ee364a/lectures.html. 2023.

[6] Shah P. Chandrasekaran V. "Relative entropy optimization and its applications". In: *Mathematical Programming* 161 (2017), pp. 1–32. DOI: 10.1007/s10107-016-0998-2.

[7] Toby Cubitt. *Quantinf package for Matlab*. http://www.dr-qubit.org/Matlab_code.html.

[8] Steven Diamond and Stephen Boyd. "CVXPY: A Python-embedded modeling language for convex optimization". In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[9] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (Jan. 2017), 295–320. ISSN: 1095-7200. DOI: 10.1137/15m1020575. URL: http://dx.doi.org/10.1137/15M1020575.

[10] Hamza Fawzi and Omar Fawzi. "Efficient optimization of the quantum relative entropy". In: *Journal of Physics A: Mathematical and Theoretical* 51.15 (2018), p. 154003. DOI: 10.1088/1751-8121/aab285. URL: https://doi.org/10.1088\%2F1751-8121\%2Faab285.

[11] Hamza Fawzi, James Saunderson, and Pablo A. Parrilo. "Semidefinite Approximations of the Matrix Logarithm". In: *Foundations of Computational Mathematics* 19.2 (2018), pp. 259–296. DOI: 10.1007/s10208-018-9385-0. URL: https://doi.org/10.1007\%2Fs10208-018-9385-0.

[12] Henrik A. Friberg. "Projection onto the exponential cone: a univariate root-finding problem". In: *Optimization Methods and Software* 38.3 (2023), pp. 457–473. DOI: 10.1080/10556788.2021.2022147. eprint: https://doi.org/10.1080/10556788.2021.2022147. URL: https://doi.org/10.1080/10556788.2021.2022147.

[13] Benyamin Ghojogh et al. *KKT Conditions, First-Order and Second-Order Optimization, and Distributed Optimization: Tutorial and Survey*. 2021. arXiv: 2110.01858 [math.OC].

[14] Francois Glineur. *Conic optimization: an elegant framework for convex optimization*. 2001.

[15] Michael Grant and Stephen Boyd. *CVX: Matlab Software for Disciplined Convex Programming, version 2.1*. http://cvxr.com/cvx. Mar. 2014.

[16] Michael Charles Grant. "Disciplined Convex Programming". PhD thesis. Stanford, 2004.

[17] Mateusz T. Madzik et al. "Precision tomography of a three-qubit donor quantum processor in silicon". In: *Nature* 601.7893 (Jan. 2022), pp. 348–353. DOI: 10.1038/s41586-021-04292-7. URL: https://doi.org/10.1038/s41586-021-04292-7.

[18] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge, England: Cambridge University Press, Dec. 2010.

[19] Lloyd N. Trefethen. "Is Gauss Quadrature Better than Clenshaw–Curtis?" In: *SIAM Review* 50 (2008), pp. 67–87.

[20] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[21] Xiao Xue et al. "Quantum logic with spin qubits crossing the surface code threshold". In: *Nature* 601.7893 (Jan. 2022), pp. 343–347. DOI: 10.1038/s41586-021-04273-w. URL: https://doi.org/10.1038/s41586-021-04273-w.

[22] Yinyu Ye. *Semidefinite Programming and Universal Rigidity*. Rigidity Microworkshop, Cornell. 2010.