OpenQKDSecurity Version 2 User Guide

John Burniston, Scott Johnstun, & Lars Kamin

June 20, 2024

Contents

Chapter 1

Introduction and Installation

OpenQKDSecurity is a software package based in MATLAB that allows users to calculate key rates for quantum key distribution (QKD) protocols using the Winick et al. framework [1] (See also Appendix A). It is extensible, allowing for user-defined protocols to be implemented, and modular, allowing for users to change specific aspects of a protocol. Our software can be used to interface with experimental data, demonstrate the theoretical scalability of protocols in various conditions, and optimize parameters to maximize key rate. Its modular structure helps break down the colossal task of numerically calculating key rates into small areas that require only domain specific knowledge. Therefore, no single person must be an expert in all areas.

1.1 External resources

We expect all users of this software package to have a baseline familiarity with MATLAB, QKD, quantum information, and convex optimization. Here are some resources to help brush up on these:

- General quantum information [2, 3, 4].
- Introduction to QKD [5] and our numerical framework [1].
- Stephen Boyl's book for convex optimization [6] and CVX's tutorial page.
- An introduction to MATLAB from MathWorks can be found here.

Matlab is a bit of an odd language with a few quirks and some useful features. Of these, we encourage you to familiarize yourself with the following:

- Every numerical data type (and object) in MATLAB is (at minimum) a 2D array, including scalar (1×1) and vectors ($n \times 1$ or $1 \times n$). Furthermore, indexing in MATLAB starts at 1.
- We make heavy use of cell arrays and structures to help manage data storage and flow.

- We also make occasional use of objects as data structures. However, their usage is largely in the lowest layers of the software, so a very basic knowledge of object oriented programming in any language is sufficient.
- If you write click on a function or class, you can view its code or documentation.
- The variable explorer is very useful for debugging purposes.

1.2 Installation

Before installing the software, ensure you have the latest version of MATLAB installed on your machine. Our software requires *at least version 2020b* for full functionality, but installing the latest version is preferable.

To install OpenQKDSecurity, download the repository on GitHub as a zip file and unzip to a preferred directory. This can be done on the main page of the repository by pressing the green “Code” button at the top of the page and then selecting “Download ZIP”. Our software has the following dependencies for its default settings:

- CVX v2.2, a library for solving convex optimization problems in MATLAB.
- QETLAB *above* v0.9, a MATLAB toolbox for operations involving quantum channels and entanglement. Note that you cannot use the version from their website as it has a bugs associated with implementing Choi matrices. *You must use their latest copy on GitHub above v0.9.* At the time of writing this means downloading their code with the green “code” button and *not* the v0.9 release.
- ZGNQKD solver (optional). An alternative to our Frank-Wolfe solver for quantum relative entropy. Currently, it only supports equality constraints. Download the zip and the ”Solver” folder and sub-folders to your path.
- MOSEK (optional), a more advanced semidefinite programming (SDP) solver than the default (SDPT3) used in CVX. Note that the MOSEK solver can be downloaded together with CVX, but requires a separate license to use. See this page for more information.

Please refer to the documentation of each of these software packages for installation instructions.

Before running OpenQKDSecurity, ensure that our software and the above dependencies are on your MATLAB path. To place a folder on your path, navigate to it in MATLAB, then right-click and select “Add to Path>Selected folder and Subfolders”. Make sure you do this for OpenQKDSecurity, QETLAB and CVX.

Once this has been done, run `cvx_setup` in the command line to ensure that CVX is set up and working properly. This command will output a list of available SDP solvers, which can be used to verify the correct installation of the above packages and the correctness of the MATLAB path.

Before you close MATLAB, go to “Home>Environment>Set Path” and click “save” at the bottom. If you don’t, then you have to add everything to your path each time you restart MATLAB.

We strongly encourage you to run `testInstall.m` *at this point to check for basic installation issues.*

Chapter 2

Software Layout

2.1 Basic Layout

OpenQKDSecurity is structured as sets of interchangeable modules that run in sequence. Each module requests a set of parameters, which may be flagged as required or optional, and adds new parameters to a larger parameter set that is used in the final key rate calculation. When receiving parameters as input, a module performs checks to validate the incoming data and ensure that important constraints are satisfied, raising an exception if any are violated. For example, we might require that the transmittance coefficient of a channel be within 0 and 1. Once a module completes execution, the parameters that it produced are merged with the larger parameter list; in the case of naming conflicts, new parameters replace old parameters. See Fig. 2.2 and Table 2.1 for an overview of the modules, their intended audience, and how they interact with each other.

At the highest level, we expect a typical user to run a protocol as laid out in `mainExample.m` found in the root directory of the package. This code block bellow represents a simple asymptotic BB84 protocol.

```
% Qubit BB84 prepare and measure. We use schmidt decomposition to reduce
% the dimension of Alice's state for enchanced speed and stability.
qkdInput = BasicBB84Alice2DPreset();

%run the QKDSolver with this input
results = MainIteration(qkdInput);

%save the results and preset to a file.
save("BasicBB84Alice2DResults.mat","results","qkdInput");

%% plot the result
QKDPlot.simple1DPlot(qkdInput,results)
```

An explanation of this code block is as follows.

1. We select and execute the preset file to be used, which, in this case, is a basic qubit BB84 prepare-and-measure protocol. The output of this function is a `QKDSolverInput` object which contains all the choices for the modules, options, and initial parameters.
2. We run the protocol specified by `qkdInput` through the call to `MainIteration`. The result, which is stored in the `results` variable, is a structure array containing the computed key rate, debug information, and information about the parameters used for that point in the scan.
3. We save the `results` struct and `qkdInput` object to a file.
4. We call the built-in `QKDPlot.simple1DPlot` function, which automatically detects scan parameters specified in `qkdInput` and plots them in the style specified.

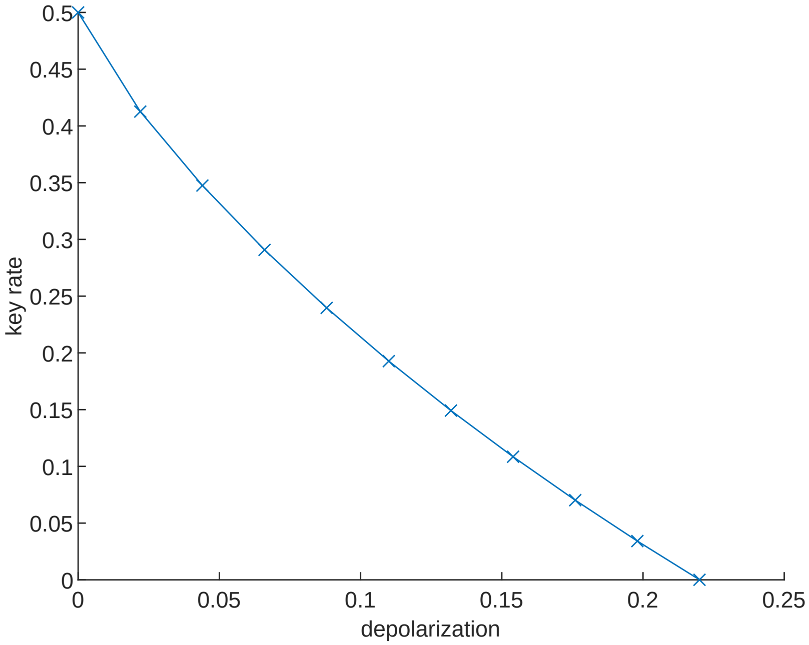


Figure 2.1: The plot that results from the simple calculation that the software is set to run when first installed. This figure presents a plot of the key rate of a qubit BB84 protocol with no loss.

Once the calculation has finished, the program will display a plot of the results, which should look like the one pictured in Fig. 2.1. Further more, the `QKDPlot.plotParametersFromFiles` and `QKDPlotplotParameters` functions can be used to compare different runs.

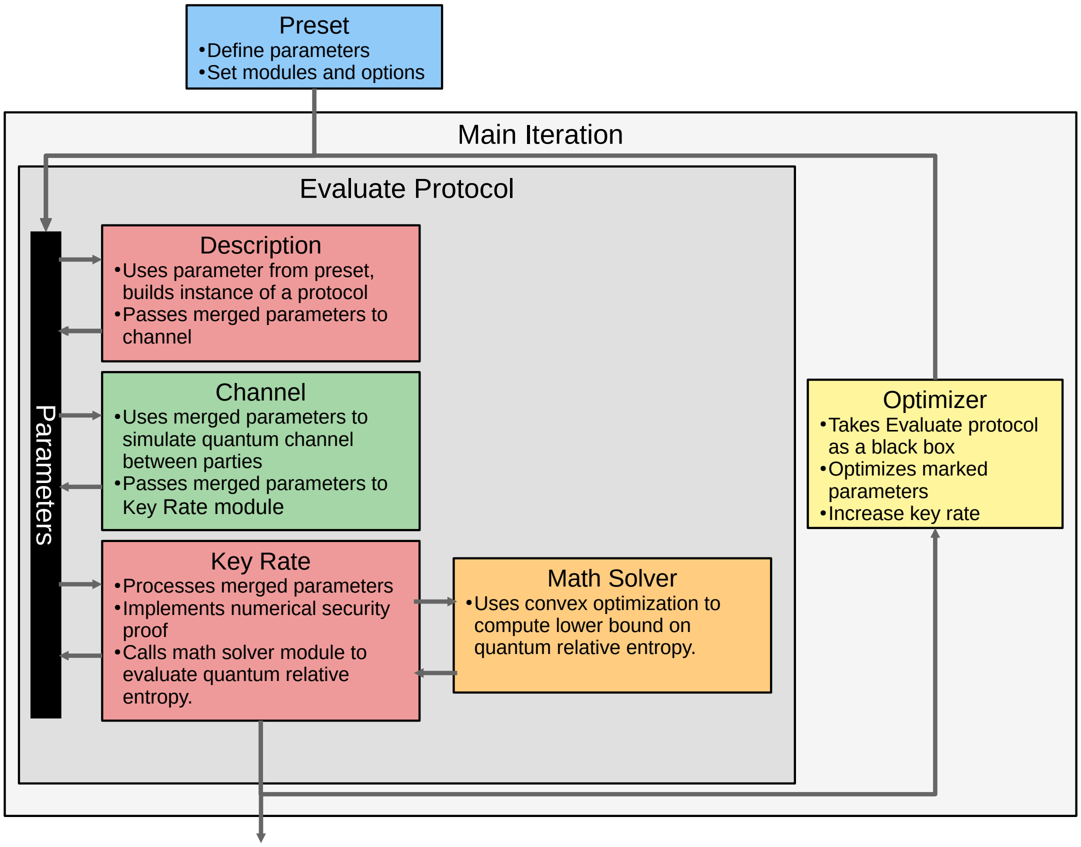

Preset
•Define parameters
•Set modules and options
Main Iteration
Evaluate Protocol
Parameters
Description
•Uses parameter from preset, builds instance of a protocol
•Passes merged parameters to channel
Channel
•Uses merged parameters to simulate quantum channel between parties
•Passes merged parameters to Key Rate module
Key Rate
•Processes merged parameters
•Implements numerical security proof
•Calls math solver module to evaluate quantum relative entropy.
Math Solver
•Uses convex optimization to compute lower bound on quantum relative entropy.
Optimizer
•Takes Evaluate protocol as a black box
•Optimizes marked parameters
•Increase key rate

Figure 2.2: Simplified layout of modules

Layer	Difficulty	Audience	Description
Preset	Basic to Intermediate	Everyone	The Preset sets up a QKD protocol by defining parameters, options and modules to use. Use cases range from simply picking a predefined preset and tweaking a parameters to selection of modules and technical options.
Description	Intermediate	Users with intermediate knowledge of QKD protocols, including proof techniques.	Using the initial parameters given in the preset, a description module constructs complete description of a QKD protocol. Measurement observables, announcement structure, and the key map are typically defined on this layer.
Channel	Intermediate	Users with intermediate to advanced knowledge of channel simulation or computing key rate from experimental data.	The channel module produces expectation values or observed frequencies from Alice and Bob's measurements on their exchanged states. These values can be obtained from numerical simulation or imported from an external source. Experimental data should also be loaded in this layer.
Key Rate	Advanced	Users with advanced knowledgeable of numerical proof techniques in QKD.	This layer is responsible for determining the key rate of a class of protocols based on the given parameters. Proof techniques such as decoy state analysis and squashing should be implemented in this layer. This layer is also responsible for processing input to be forwarded to the math solver.
Math Solver	Advanced	Users with advanced knowledge of convex optimization.	Calculates a strict lower bound on the quantum relative entropy between the key and Eve via convex optimization.
Optimizer	Advanced	Users with knowledge of general optimization techniques with the intent to minimize the number of calls to the objective function needed to find the global maximum.	Routines to perform unstructured optimization on selected parameters, using the key rate as the objective. This layer is given a wrapped version of the protocol with options modified to reduce precision but decrease calculation time.

Table 2.1: Overview of preset and module types used by the software. See Chapter 3 for more details.

2.2 Detailed Layout

This section is for those who want a more detailed description of how modules interact. Here we walk through the major components of the functions `MainIteration` and `EvaluateProtocol`.

2.2.1 MainIteration

The heavy lifting of the key rate calculation is facilitated by the `MainIteration` function. Its job is to parse parameters, set up individual runs of the protcol (based on said parameters), then gather and report key rates, debug information, and parameter selections. The diagram of `MainIteration` in Fig. 2.3 represents the following steps:

1. A preset is run which compiles a set of initial parameters, modules and their options, and a list of global options into a `QKDSolverInput` object.
2. The new QKDSolverInput object is given to the `MainIteration` function and executes the preset.
3. `MainIteration` loops over all combination of scan parameters and performs the following:
    (a) `MainIteration` sets the current parameters to the fixed parameters plus the current selection of scan parameters.
    (b) A wrapped version of the protocol which only requires the optimize parameters as input is constructed.
    (c) The optimization module is given the wrapped function handle, the names, bounds and initial values of the optimization parameters, and its options.
    (d) The optimization module performs an unstructured global maximization to improve key rates by tweaking the optimization parameters.
    (e) The best choice of optimization parameters is returned and merged into the current parameters.
    (f) The protocol is evaluated again with different options (typically to allow for more precision).
    (g) The current parameters, key rate lower bound and debug information is appended to the results.
4. The results structure is returned.
5. The preset and results are saved to a file and a plot of key rate vs scan parameter is produced.

2.2.2 EvaluateProtocol

`MainIteration` relies on the function `EvaluateProtocol`, which facilitates a key rate calculation based on the parameters assembled and selected in `MainIteration`. Its main job is to coordinate the flow of parameters from the description module to the key rate calculation, report the key rate, and gather individual modules' debug information. Fig. 2.4 represents the following steps for `EvaluateProtocol`:

1. The description options are merged with the global options and are passed to the description module with the current parameters.

2. The description module generates new parameters and debug information.
3. The new parameters are merged with the old parameters.
4. The same steps apply for the channel module.
5. Global options are merged with the key rate options and (separately) the global options are merged with the math solver options.
6. The parameters, key rate module options, math solver module options, and a function handle for the math solver are passed to the key rate module.
   (a) The key rate module performs performs any additional coarse graining, calculates error correction cost and applies proof technique steps such as decoy analysis.
   (b) The key rate module constructs input parameters for the math solver module to describe the constraints and objective function for the relative entropy minimization.
   (c) The key rate module runs the math solver module (possibly multiple times depending on the complexity of the protocol).
   (d) The math solver module returns a safe, guaranteed, lower bound on the relative entropy between the key and Eve, alongside debug information.
   (e) The key rate module performs any outstanding steps and calculates the key rate.
7. The key rate lower bound and all debug information is compiled and returned.

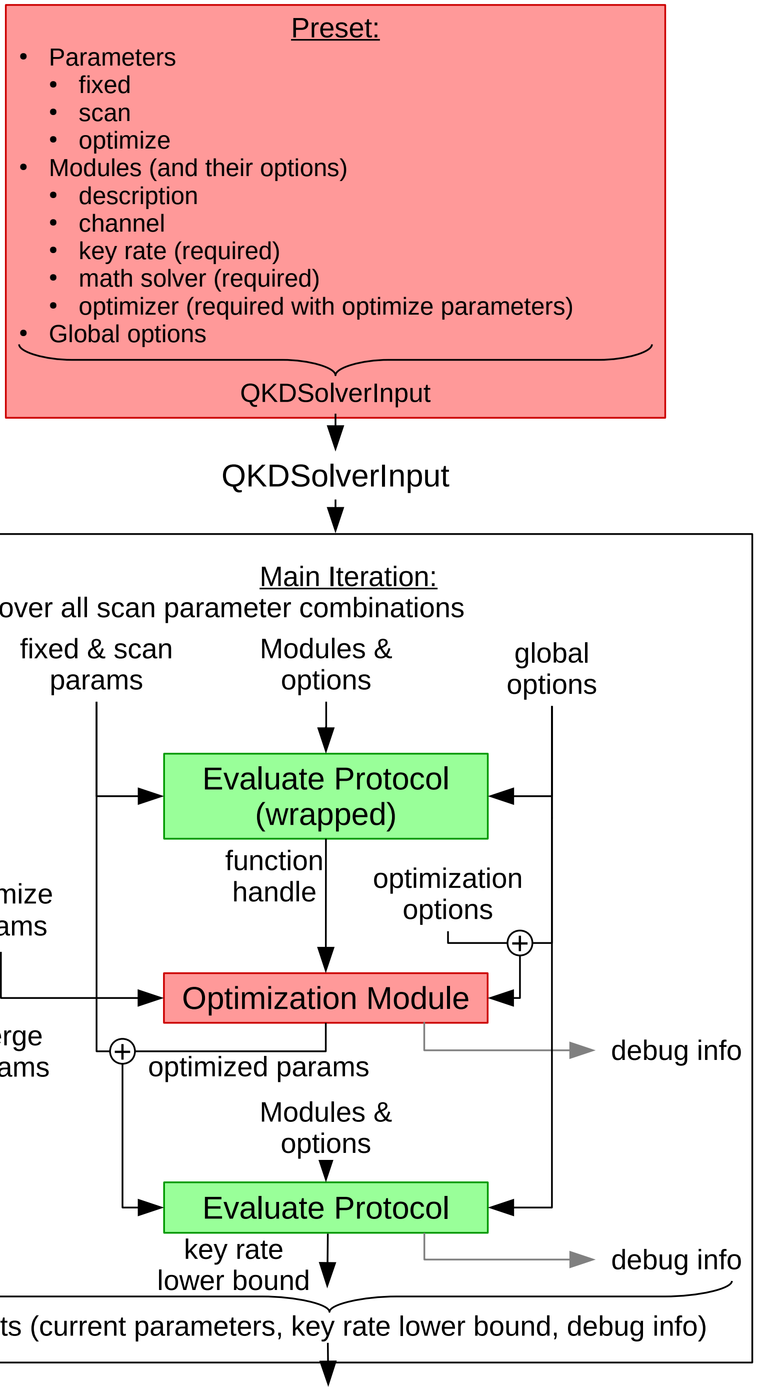


Figure 2.3: A breakdown of evaluate protocol can be found in Fig. 2.4.

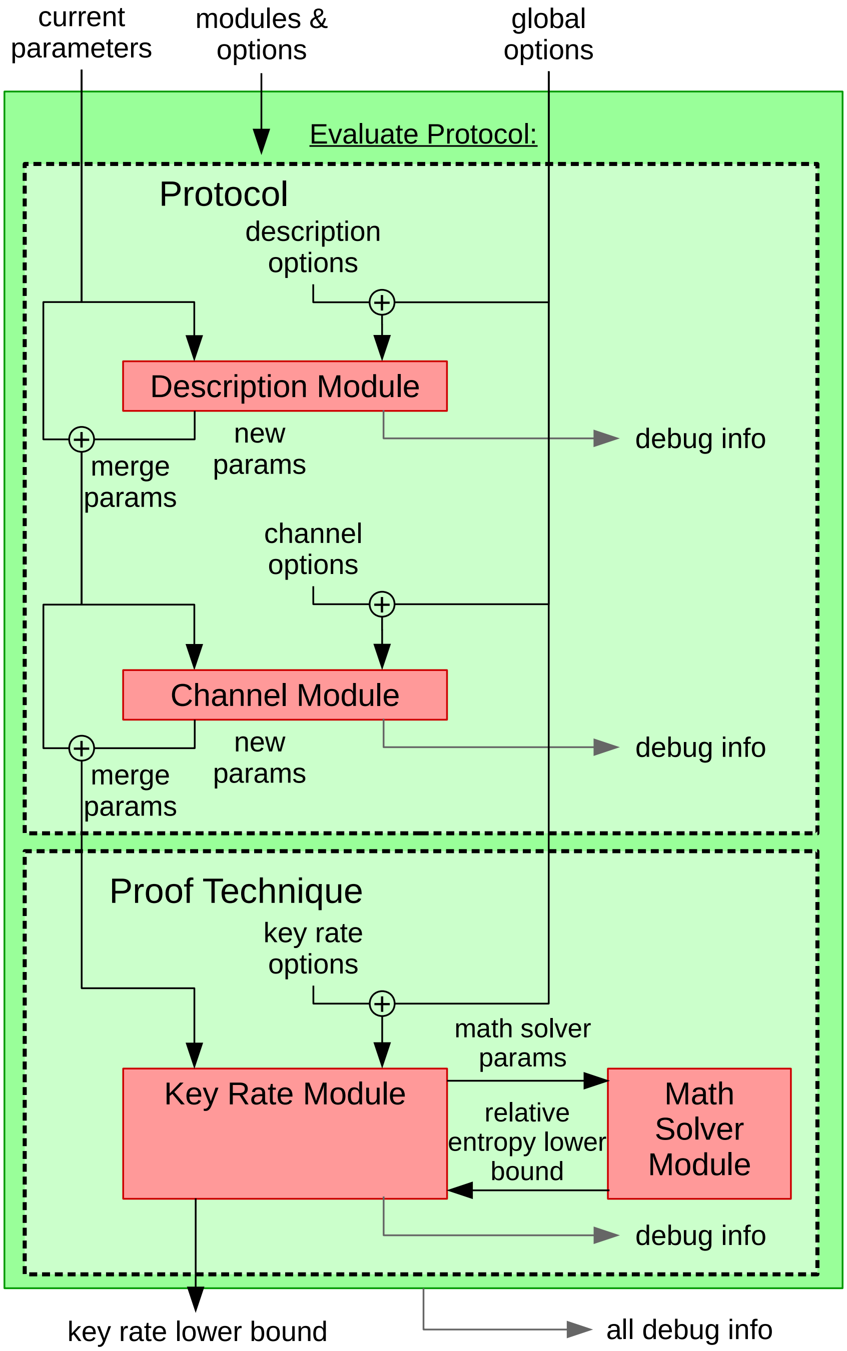


Figure 2.4: The math solver module's options and debug information is not included for clarity.

Chapter 3

Presets and Modules

3.1 Preset

The first layer of customization that the program offers is modifying preset files. Preset files are found in the folder pertaining to a protocol in the `BasicProtocols` directory. Preset files define the following parts of a protocol:

- Parameters for the protocol, such as basis choice probabilities, transmission efficiencies, magnitudes of various sources of noise, and security tolerances
- Modules that must be run to supply or compute vital information in the protocol. Typically, these will be the description module, channel module, key rate module, math solver module, and (optionally) optimizer module.
- Global options, such as error handling modes, the back end solver used in CVX, and verbosity.

The preset file generates a `QKDSolverInput` object called `qkdInput`, which is stored in the `results` struct at the end of a run of the software. All the parameters, options, and modules needed for the protocol are stored in this object. This allows users to save the input parameters of one run of the software and use them for another run.

3.1.1 Adding Parameters to Presets

Parameters are tuneable values that affect the statistics and calculations in a protocol. Parameters come in three types:

- **Fixed** parameters are specified as a single value and remain constant throughout a key rate calculation.
- **Scan** parameters are given as a collection of values to take on. This is implemented via a cell array. At runtime, the software takes each value specified in the cell array and uses it to compute a key rate at that point. Scan parameters are used to generate key rate plots. You may use multiple scan parameters and each combination between the multiple parameters will be used.

Function name	Arguments	Description
`addFixedParameter`	`name` (string) `value` (anything)	Adds a fixed parameter to the input structure
`addScanParameter`	`name` (string) `values` (cell array)	Adds a scan parameter to the input structure
`addOptimizeParameter`	`name` (string) `value` (struct)	Adds an optimize parameter to the input structure. The `value` struct must have fields `initVal`, `upperBound`, and `lowerBound`
`removeFixedParameter`	`names` (string array)	Removes a list of fixed parameters from the input structure
`removeScanParameter`	`names` (string array)	Removes a list of scan parameters from the input structure
`removeOptimizeParameter`	`names` (string array)	Removes a list of optimize parameters from the input structure

Table 3.1: Basic methods to add, set and remove parameters for the `QKDSolverInput` object.

- **Optimize** parameters are given as a structure with three fields, `initVal`, `upperBound`, and `lowerBound`, which define the initial value, upper bound, and lower bound of the parameter in question respectively. At each scan iteration, the parameter will be initialized to `initVal` and then be optimized using a built-in or external optimizer, bounded within the range set by `upperBound` and `lowerBound`. This is accomplished by performing many rounds of key rate calculation at larger tolerances than the main calculation.

The `QKDSolverInput` class has several methods to set, add and remove the initial parameters. We present them in table Table 3.1. Note when using any of the "add X Parameter" functions, if you add a parameter that shares a name with a previously added parameter, then the original parameters is removed and the new one is added.

For example, the following block of code adds three parameters: the quantum channel misalignment angle θ, named `misalignmentAngle`, as a fixed parameter with value $\frac{\pi}{20}$; the quantum channel transmissivity η, named `eta`, as a scan parameter with values ranging from 0.1 to 1 with a step size of 0.1; and the basis choice probability for the z-basis p_z, named `pz`, as an optimize parameter with upper bound 0.99, lower bound 0.5, and initial value 0.75.

```
% Add misalignment angle as fixed parameter
qkdInput.addFixedParameter('misalignmentAngle', pi/20);
% Add channel tranmissivity as a scan parameter
qkdInput.addScanParameter('eta', num2cell(0.1:0.1:1));
% Create struct for optimization
pzOptimStruct.initVal = 0.75;
pzOptimStruct.lowerBound = 0.5;
pzOptimStruct.upperBound = 0.99;
qkdInput.addOptimizeParameter('pz', pzOptimStruct);
```

Parameter Naming Conventions

Not just any string can be used to define a parameter name. As field names of structs, they must be valid MATLAB variable names. We further restrict this to variables that only use alphanumeric symbols (a-z, A-Z, and 0-9). We restrict the use of the ‘_’ symbol for special parameters which are handled differently. For example, `eta` and `randomVariable1A` are valid variable names, but `1NormLimit` and `simple_distribution` are not.

Special parameters follow a slightly altered naming convention. They are comprised of 3 substrings separated by underscores in the following order:

- parameter type: All caps code for the type of special parameter.
- parameter name: The name for the parameter. It must follow the default parameter naming conventions.
- numeric code: A string of digits from 0 to 9.

As such, a special parameter would look like `SPECIALPARAMETERTYPE_parametername_numericCode`. Currently, the only special parameter type is the GROUP type which bundles any tagged parameter with the same name together in a cell array sorted by the numeric code. This lets one add different parts of a single parameter into different initial parameter types as in the following code block.

```
qkdInput.addScanParameter("GROUP_mu_1",{0.5});
qkdInput.addFixedParameter("GROUP_mu_2",0.9);
qkdInput.addFixedParameter("GROUP_mu_3",0);
% In EvaluateProtocol, this converts to the parameter
% params.mu = {0.5,0.9,0};
```

Caution 3.1.1: Name Overwriting

- If there is a scan parameter named `mu`, then adding fixed parameter named `mu` will remove the scan parameter. The same is true for any combination of scan, fixed and optimize parameters.
- A parameter with name `mu` will be overwritten by a parameter named `GROUP_mu_1` after it is converted.

3.1.2 Selecting Modules for Presets

In this section of the preset file, we define the modules to be used in our protocol. Modules are the working components of the QKD protocol, setting up important pieces like the measurement observables, Kraus operators for the $\mathcal{G}$ and $\mathcal{Z}$ maps, defining and implementing the quantum channel action, optimizing parameters, and calculating key rates. Modules in our software are MATLAB functions that follow specific rules for input and output. The available modules are **description**, **channel**, **keyRate**, **optimizer**, and **mathSolver**. A more extensive explanation of the specific modules used in our software will be given in Section 3.2.

In the preset file, we tell the software which modules should be used for our QKD protocol. We use function handles to point to the function representing the specific module. Each module is specified by an object which holds the module function as well as options to use in the module. Most modules included in the software do not require extra options; typically, only the math solver module requires a list of options. For example, to assign the file `BB84DescriptionFunc.m` to your protocol's description module, we could run the following lines of code:

Function name	Arguments	Description
`setDescriptionModule`	`moduleFunction` (QKDDescriptionModule)	Sets the module used for the protocol description.
`setChannelModule`	`channelModule` (QKDChannelModule)	Sets the module used for the channel model.
`setKeyRateModule`	`keyRateModule` (QKDKeyRateModule)	Sets the module used for computing key rate.
`setOptimizerModule`	`optimizerModule` (QKDOptimizerModule)	Sets the module used for optimizing parameters.
`setMathSolverModule`	`mathSolverModule` (QKDMathSolverModule)	Sets the module used for relative entropy calculations.

Table 3.2: List of functions to set modules from the `QKDSolverInput` object.

```
descriptionModule = QKDDescriptionModule(@BB84DescriptionFunc);
qkdInput.setDescriptionModule(descriptionModule);
```

The first line uses the constructor of the `QKDDescriptionModule` class, which takes in a function handle and (optionally) a struct for options and a struct for options that only override when the software is optimizing a parameter. Since `BB84DescriptionFunc.m` does not require options, we can simply pass the function handle into the constructor and the object can be created. We then use the `setDescriptionModule` function to set `BB84DescriptionFunc` as the description module for our protocol.

To demonstrate the inclusion of options into a module, we next show an example of setting the math solver module.

```
1 mathSolverOptions.initmethod = 1;
2 mathSolverOptions.maxiter = 10;
3 mathSolverModule = QKDMathSolverModule(@FW2StepSolver, mathSolverOptions);
4 qkdInput.setMathSolverModule(mathSolverModule);
```

In the first two lines, we set up the options-storing struct `mathSolverOptions`, which contains some fields and values relevant to the math solver. In line 3, we use the `QKDMathSolverModule` constructor to assign a function and options to a module object. Finally, in line 4, we set the math solver module to the module that was just created.

The `QKDSolverInput` class has functions that take in particular modules for use in a QKD protocol. You can find some of the most relevant ones in in Table 3.2.

3.1.3 Global Options

The final function a preset file can perform is defining the global options used in the current execution of the software. These are higher-level options that are held constant across scanning and optimization.[1] A module may also give

[1] The constructor for each module has additional struct inputs `options`, `optimizerOverrideOptions`, which can hold options that override the global options and global options while optimizing a parameter.See Sections 3.1.2 and 3.2.3

Option	Possible values	Default value	Description
`cvxSolver`	“SDPT3”, “MOSEK”, etc. (any locally installed solver)	“SDPT3”	Defines the base solver used for any calculations that use CVX in a protocol. Note that this option can be overridden locally by functions that perform convex optimizations. See the CVX solver webpage for a list of available solvers and download links.
`cvxPrecision`	“Low”, “Medium”, “Default”, “High”, “Best”	“High”	Sets the numerical tolerance allowed in SDPs solved by CVX. See the CVX page on controlling precision for more detailed information.
`verboseLevel`	0, 1, 2	1	Specifies the amount of information sent to the Command Window during the execution of the protocol. 0: Display nothing but iteration step. 1: Display high level information about modules, math solver calculations, and the key rate. 2: Display everything, including all underlying CVX output.
`errorHandling`	0, 1, 2	2	Defines the protocol for handling errors encountered in the software. 0: Catch the error, set key rate for this iteration to zero, and continue calculating the next point. No warning is issued. 1: Same as 0, but warns the user whenever an error is caught. 2: Catch the error, halt program execution, and print stack trace to the Command Window.

Table 3.3: List of global options.

structs that for details. The available global options, with their descriptions, are found in Table 3.3. Here is a basic example of how to set specific global options.

```
globalOptions = struct();
globalOptions.errorHandling = ErrorHandling.DontCatch;
globalOptions.cvxSolver = "Mosek";
qkdInput.setGlobalOptions(globalOptions);
```

3.2 Modules

As briefly described in Section 3.1.2, modules are the major working components of QKD protocols. The available modules are **description**, **channel**, **keyRate**, **optimizer**, and **mathSolver**, and a brief overview of each can be found in Table 2.1. In this section, we will go over the commonalities shared by nearly every module before going into specific module types.

At a high level, modules function by taking in a list of current parameters, options and a `DebugInfo` object (Section 3.2.4). The module then checks for the parameters and options it requires, validates the value and begins its task. After it is finished, the module will return either a list of new parameters to merge with the old list, or a very special value such as a lower bound on relative entropy or the key rate. Additionally, the `moduleParser` used to select and validate objects will also be returned to aid in some internal processes. To help illustrate this point, below is the base function signature most module types use.

```
function [newParamsOrVeryImportantValue, modParser]...
  = nameModuleTypeFunc(params,options,debugInfo)
% detailed description of module
arguments
    params (1,1) struct
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end
% parse options
% parse parameters
% Do task.
end
```

Furthermore, we provide an example of a complete module including the layout of its documentation. It illustrates the major steps of parsing options, parsing parameters, and utilizing them. It also shows how to add information to the `DebugInfo` object, add new parameter, and best practices for sending messages to the console.

Listing 3.1: Simple module function example.

```
function [newParams, modParser]= simpleModuleFunc(params,options,debugInfo)
% A simple module to show how they are constructed. This one calculates the
% affine transformation y = Mx+b for a vector x, matrix M, and non-negative
% offset b. Because it is very important, the module also allows the user to
% change M to its complex conjugate. Note, some module types have a
% slightly different function signature. Here, let's call it a channel
% module.
%
% Input Parameters:
% * M: n x m matrix to multiply x by. It can be real or complex.
% * x: m x 1 column vector multiplied by M. It can be real or complex. The
%   dimensions must be compatible for matrix multiplication.
% * b (0): n x 1 column vector or scalar. Provides an optional offset to
%   turn this from a linear transformation to an affine transformation. All
%   components of b must be non-negative.
% Output Parameters:
% * y: The result of the calculation y = Mx+b.
% Options:
% * verboseLevel (global option): See makeGlobalOptionsParser for details.
% * useConjugate (false): When set to true, the calculation swaps M with
%   its complex conjugate (not complex conjugate transpose). The new
%   calculation is thus y = M^*x+b.
% Debug Information:
% * mx: The very important linear transformation part. Useful for checking
%   the operation before the affine offset.
%
% Note that the Matlab "see also" must be the last line, and can't span
% several lines.
%
% See also QKDChannelModule, conj, makeGlobalOptionsParser

arguments
    params (1,1) struct
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end

%% Options parser

% All options parsers should start with this line
optionsParser = makeGlobalOptionsParser(mfilename);

optionsParser.addOptionalParameter("useConjugate", false, @islogical);
optionsParser.parse(options);
options = optionsParser.Results;

%% Module parser

modParser = moduleParser(mfilename);

% Matrix we want to multiply by:
modParser.addRequiredParam("M", @(x) mustBeNumericOrLogical(x));
% Make sure it's 2 dimensional:
modParser.addAdditionalConstraint(@ismatrix, "M");

% Vector we want to multiply:
modParser.addRequiredParam("x",@(x) mustBeNumericOrLogical(x));
% Make sure it's a column vector:
modParser.addAdditionalConstraint(@iscolumn, "x");

```

```
% Make sure M and x are compatible for multiplication:
modParser.addAdditionalConstraint(@(M,x) size(M,2)==size(x,1), ["M","x"]);

% Non-negative vector we want to add:
modParser.addOptionalParam("b", 0, @(x) mustBeNonnegative(x))
% Make sure it's a column vector:
modParser.addAdditionalConstraint(@iscolumn, "b");

% Make sure M and b are compatible for addition:
modParser.addAdditionalConstraint(@(b,M) isscalar(b)...
    || isequal(size(b,1),size(M,1)),["b","M"]);


modParser.parse(params); % Apply the parsing scheme.

params = modParser.Results; % Override the parameters with parsed ones.
% From this point on, DO NOT TOUCH THE PARSERS.

%% The actual function

if options.useConjugate % Swap to the complex conjugate of M when requested.
    params.M = conj(params.M);
end

% The linear part is very important. So we calculate it first and make sure
% we store it in the debug information.
mx = params.M*params.x;

% Some information for debugging.
% Store immediately in case an error is thrown in the next step.
debugInfo.storeInfo("mx",mx);

% We further display on console if the verbose level is set high enough.
if options.verboseLevel >= 1
   disp("The linear part of the transformation is:");
   disp(mx);
end

y = mx+params.b;

% Add the parameter y to the returned parameters
newParams.y = y; % Make sure it's newParams and not params.

end
```

3.2.1 Adding New Parameters

Before we get into module parsers and debug information, let's have a quick look at how new parameters are added. If we want to add a new parameter named `smallVector`, and assign it the array `[1,2]`, then we would write in the main body of the function

```
newParams.smallVector = [1,2];
```

This adds the field `smallVector` to the struct `newParams`. When the module's main function returns, `EvaluateProtocol` will merge the new parameters from `newParams` into the original parameters passed in. For example if the original parameters were given by the struct,

```
params.smallValue = 0.9;
params.foo = "foo";
params
%  struct with fields:
%
%    smallValue: 0.9000
%           foo: "bar"
```

then after we run the module that adds `smallVector`, the new parameters would be

```
params
%  struct with fields:
%
%    smallValue: 0.9000
%           foo: "bar"
%   smallVector: [1 2]
```

If a new parameter shares a name with an old parameter, then when the new and old parameters are merged together, the new parameter overwrites the old and a warning is given.

Caution 3.2.1: Overwriting newParams

A common mistake is writing a new parameter, say `newParams.smallVector =1;`, then further down in the same module accidentally overwriting it with `newParams.smallVector = 10;`. This is just basic struct behaviour in MATLAB and it will not prompt a warning.

3.2.2 Module Parsers and Requesting Parameters

Module parsers are built from and behave similarly to Matlab's built in inputParser. Module parsers serve two main purposes:

1. Declare the required and optional parameters a module requests.
2. Filter and validate the parameters passed to the module with a "look before you leap" defensive coding practice.

`Function name`	Arguments	Description
`moduleParser`	FunctionName (string)	Constructor for a new module parser. Please use the name of the file. This can be easily obtained with the `mfilename` function.
`addRequiredParam`	obj (ModuleParser) paramName (string) validationFunc (function_handle)	Adds a required parameter to the ModuleParser with the given name. Optionally a validation function may be included.
`addOptionalParam`	obj (ModuleParser) paramName (string) defaultVal (anything) validationFunc (function_handle)	Adds an optional parameter to the ModuleParser. A default value for the optional parameter must be provided alongside the input. Optionally, a validation function may be included.
`addAdditionalConstraint`	obj (ModuleParser) validationFunc (function_handle) paramNames (string array)	Adds extra validation functions to be run on the named parameter. If multiple parameter names are given, it will attempt to call the validation function with all the arguments listed in the same order as paramNames. (Good for cross-argument constraints.)
`parse`	obj (ModuleParser) inputParams (struct) warnUnusedParams (logical)	Applies the constructed ModuleParser scheme to inputParams. Throws an error if any condition is violated or if any required parameters are not present. Optionally, the name value argument “warnUnusedParams” (default false) can be set to true and the user will get a warning if unused parameter were supplied.

Table 3.4: ModuleParser Methods

Property	Description
FunctionName	Name given to the input parser which it uses in error reports.
Results	Struct with the name value pairs of all matched arguments and default values for unmatched optional arguments.
UsingDefaults	Cell array containing the names of all the parameters that were not found and are using their default value.
Unmatched	Struct containing the name value pairs of inputs that could not be matched to any found in the ModuleParser’s required or optional parameter lists.
Parameters	Cell array of all the parameter names the ModuleParser uses.

Table 3.5: ModuleParser Public Properties. All are read-only and are identical to the properties found in Matlab’s built in inputParser.

A brief summary of a module parser's methods and properties can be found in Tables 3.4 and 3.5. Note that the names of parameters added to module parsers must be exact matches, and are case sensitive.

Let's look back at listing 3.1 for a basic implementation. For now, we'll skip the options parser and focus on just parsing the module's parameters. We create a new `moduleParser` object with the line

```
modParser = moduleParser(mfilename);
```

This sets up a new blank module parser. The function `mfilename` gives the name of the current function to the module parser. If an error is thrown by the parser, then the message will use the module's function name to make it clearer where the error occurred. This function has two required inputs, a matrix M and a vector x which are multiplied together. We can simply add them by writing

```
modParser.addRequiredParam("M");
modParser.addRequiredParam("x");
```

but this would place no restrictions on the values the user could pass in. To help validate parameter values, we can use functions that either return logical values such is `ismatrix` (returns true if the check passes), or return nothing and throw an error if the test fails such as Matlab's argument validation functions. For example, we would like to restrict M and x to be numeric types (or logical which is automatically converted). We can use the argument validation function `mustBeNumericOrLogical` to achieve this, by writing

```
modParser.addRequiredParam("M",@(x) mustBeNumericOrLogical(x));
modParser.addRequiredParam("x",@(x) mustBeNumericOrLogical(x));
```

Caution 3.2.2: Argument Validation Functions in Module Parsers

Matlab's argument validation functions like `mustBePositive` don't just throw an error, they throw an error from the calling function. This can lead to some more confusing stack traces. We recommend these argument validation functions should be wrapped in a short anonymous function when added to module parsers. For example,`@(x) mustBePositive(x)`.

This fixes the data type, but it doesn't place any other limits on the variables. We can only add one validation function to each call of the `addRequiredParam` method, so we need to use additional constraints. After we've added M and x as parameters to the module parser, we can use the `addAdditionalConstraint` method apply further checks. This method takes in a single function, and an array of strings to fill in the function inputs. For example, we can use this method to ensure that M is a matrix, x is a vector, and that M and x have compatible dimensions for matrix multiplication.

```
modParser.addAdditionalConstraint(@ismatrix, "M");
modParser.addAdditionalConstraint(@iscolumn, "x");
modParser.addAdditionalConstraint(@(M,x) size(M,2) == size(x,1), ["M","x"]);
```

For the optional affine offset b, we use the related method `addOptionalParam` which also takes a default value to assign if the parameter is not specified.

```
modParser.addOptionalParam("b", 0, @(x) mustBeNonnegative(x))
modParser.addAdditionalConstraint(@iscolumn, "b");
modParser.addAdditionalConstraint(@(b,M) isscalar(b)...
    || isequal(size(b,1),size(M,1)),["b","M"]);
```

Here we, enforce b to be a non negative scalar or a non negative vector with the same number of dimensions as output from M. By default, we set $b = 0$ which represents no affine offset. We recommend that any constraint more complex than the one between b and M above should be written as its own validation function with a descriptive error message.

With all the parameters added to the parser, we now run it and assign the results back to the `params` variable. The parameters can now be used by the function.

```
modParser.parse(params); % Apply the parsing scheme.
params = modParser.Results; % Override the parameters with parsed ones.
% From this point on, DO NOT TOUCH THE PARSERS
```

Caution 3.2.3: Module Parser Best Practices

To ensure that the module runs with expected and predictable behaviour please ensure the following:

- Do not use the `params` struct before applying the parser.
- Do not use the `moduleParser.UsingDefault` to determine which parameters were not provided. A module should not behave differently when the parameter is not given, and when the default value is passed in. Similarly, do not use the unmatched parameters in `moduleParser.Unmatched`.
- Do not reuse or assign a new module parser to `modParser`. The module parser is returned to help the system track the flow and assignment of parameters.

3.2.3 Options

Each module takes in a list of technical options to help fine tune the module's functionality such as setting numerical tolerances, toggles to run special routines or how much information the module should report. At minimum, all modules must accept (though they don't have to use) the global options from Section 3.1.3.

Unlike parameters, a module's options are defined entirely in the preset and cannot be changed outside of the following options overriding hierarchy (earlier entries are overwritten by later entries).

1. Default global options listed in Table 3.3.
2. The global options specified in the preset from Section 3.1.3.
3. The optimizer module's global override options (Only when the protocol is run in an optimization routine).
4. The module's individual options.

5. The module's optimizer override options (Only when the protocol is run in an optimization routine).

The most common use case for the optimizer module's global override options and the module's optimizer override options is to reduce the verbosity level to prevent the console being flooded and to reduce the number of steps or tolerance of expensive computations.

Setting Module Options and Optimizer Override Options

Upon constructing a module for a preset, the user can specify name-value pairs for technical options the module should use for each run. These options cannot be changed between iterations of scan parameters and optimization calls. These options are selected by creating a struct of name value pairs and passing them into the module's constructor. As can be seen in the next example, there is also a second type of module options called, optimizer override options used when performing global optimization. See Section 3.2.3 for override order and Section 3.2.9 for more on global optimization.

Listing 3.2: Setting Module Options

```
%Inside a preset

% Setting regular options
myModuleOptions = struct();
myModuleOptions.option1 = 1;
myModuleOptions.option2 = "foo";

% Setting optimzer module

% We want "bar" to override "foo" during global optimization for option2, and we
% want option1 to stay as is.
myModuleOptimizerOverrideOptions = struct();
myModuleOptimizerOverrideOptions.option2 = "bar";

% Add them to our channel model
channelModel = QKDChannelModule(@myModule,myModuleOptions,
    myModuleOptimizerOverrideOptions);
qkdInput.setChannelModule(channelModel);
```

Parsing Options

Options are passed to a module in a struct in the same way parameters are passed in. As such, we use a special `moduleParser` (we call an `optionsParser`) and add options using the `addOptionalParam` method like in Section 3.2.2. Following the example in listing 3.1, we start by using the special function `makeGlobalOptionsParser` to set up a `moduleParser` that can handle (at minimum) the global options,

```
optionsParser = makeGlobalOptionsParser(mfilename);
```

Additional module specific options are added using `addOptionalParam`, then the parser is run.

```
optionsParser.addOptionalParameter("useConjugate",false, @islogical);
optionsParser.parse(options);
options = optionsParser.Results;
```

Caution 3.2.4: Options Parser Best Practices

- Don't add required options. It defeats the purpose of making them optional.
- Run the options parser before you parse parameters. This way you can use the options when parsing parameters.
- All other best practices from caution 3.2.3 also apply here.

3.2.4 Debug Information

To help organize and store information, each module accepts a pass by reference object of type `DebugInfo`. Information in these objects are collected and returned along side the key rate at the end of the main iteration. This is used to store debug information, results from important intermediate steps, and other information a user might want to inspect later. For example, many channel modules store Alice and Bob's state ρ_{AB} before measurement results are simulated. Furthermore, even if an error occurs in a module, information stored in a `DebugInfo` object will persist so long as the global option `errorHandling` is set to 1 or 2. This allows you to inspect a module even when it's failing. Each `DebugInfo` object has access to the methods in Table 3.6, though an average user will typically only need to know `storeInfo` and `addLeaves`.

From listing 3.1 we see that the module stores the intermediate result `mx` for later inspection by running

```
debugInfo.storeInfo("mx",mx);
```

Nesting DebugInfo

For each run, debug information is stored in a recursive tree data structure. `EvaluateProtocol` generates a root `DebugInfo` object, then adds leaves named `descriptionModule`, `channelModule`, and `keyRateModule`, which it passes to the respective modules. If you need to call one module from another, or if a function requests a `DebugInfo` object, then add a new leaf (with a unique name) to your current debugInfo, and pass the leaf to the function. This will ensure each module and function gets a clean `DebugInfo` object. Here is an example with two simple functions.

Caution 3.2.5: Overwriting Debug Info

When adding leaves to a `DebugInfo` object make sure to give them unique names or they will overwrite each other.

```
baseDebugInfo = DebugInfo();

```

Method name	Arguments	Description
`DebugInfo`	none	Base constructor for a new `DebugInfo` object.
`storeInfo`	`obj` (DebugInfo) `name` (string) value (anything)	Adds a new name value pair to the debug info. Name cannot clash with the name of a leaf and must be a valid variable name.
`removeInfo`	`obj` (DebugInfo) `names` (string array)	Removes the debug information name value pairs given by the names. All names must be present or an error is thrown.
`addLeaves`	`obj` (DebugInfo) `names` (string array)	Adds new leaves (new DebugInfo objects) to the base debug info labeled with names from the array. The names must be unique and not clash with the name of any info field. Returns an array of all debugInfo leaves added this way.
`removeLeaves`	`obj` (DebugInfo) `names` (string array)	Removes the debug information leaves associated with each name. All names must be present or an error is thrown.
`DebugInfo2Struct`	`obj` (DebugInfo)	Creates a copy of the debug information as a struct with name value pairs consisting of the info names and values. The leaves are converted to structs recursively and added on with their associated names.

Table 3.6: List of methods provided by the DebugInfo class.

```
firstModuleFunc(baseDebugInfo);

% output as nested structs
compiledDebugInfo = baseDebugInfo.DebugInfo2Struct()
% struct with fields:
%
%          foo: "bar"
%    firstLeaf: [1x1 struct]
%   secondLeaf: [1x1 struct]

compiledDebugInfo.firstLeaf
% struct with fields:
%
%       bigVal: 11
compiledDebugInfo.secondLeaf
% struct with fields:
%
%       bigVal: 22


function firstModuleFunc(debugInfo)
arguments
    debugInfo (1,1) DebugInfo
end
debugInfo.storeInfo("foo","bar");

debugLeaves = debugInfo.addLeaves(["firstLeaf","secondLeaf"]);

secondModuleFunc(1,debugLeaves(1));
secondModuleFunc(2,debugLeaves(2));
end


function secondModuleFunc(inputValue,debugInfo)
arguments
    inputValue (1,1) double
    debugInfo (1,1) DebugInfo
end
debugInfo.storeInfo("bigVal",11*inputValue)
end
```

3.2.5 Description Modules

Listing 3.3: Description Module Function Signature

```
function [newParams, modParser]...
  = NAMEDescriptionFunc(params,options,debugInfo)
% detailed description of module
arguments
    params (1,1) struct
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end
%code
end
```

While the key rate module specifies a class of solvable protocols, the description module provides the details and set up for a specific protocol. As seen from Fig. 2.4, it is the first module executed by `EvaluateProtocol`, and it constructs the majority of the parameters that are called by later modules (The chief exception being measurement statistics).

The following new parameters are almost always defined in a description module:

- **POVMA**, the Positive Operator-Valued Measure (POVM) $\{M^A_{\alpha,x}\}$ that Alice uses to measure her system.
- **POVMB**, the POVM $\{M^B_{\beta,y}\}$ that Bob uses to measure his system.
- **observablesJoint**, the set of joint observables that Alice and Bob use. Typically, this will be the full set of possible tensor products between elements of $\{M^A_{\alpha,x}\}$ and $\{M^B_{\beta,y}\}$. We use the convention of Alice is indexed down (first index) and Bob across (second index).
- **rhoA**, the density operator ρ_A that describes Alice's portion of the shared state $\rho_{AA'}$.
- **krausOps**, the Kraus operators that define the $\mathcal{G}$ map, a quantum channel that records measurement results and assigns announcements and key bit registers.
- **keyProj**, the Kraus operators for the pinching channel $\mathcal{Z}$ that projects the components of ρ_{AB} onto each key bit.
- **announcementsA** and **announcementsB** arrays with the same size as `POVMA` and `POVMB` respectively. They contain string arrays that represent Alice and Bob's announcements for each of their measurement outcomes. Note that this system is limited to protocols where Alice and Bob's announcements are done simultaneously in a single round, and are deterministic.
- **keyMap**, a struct-like class that stores a list of accepted announcement pairs between Alice and Bob and a list of the corresponding key bits assigned to Alice's POVM. Once again, this is currently limited to single round, deterministic key maps.

Of the above items, only `observablesJoint` is required to be specified in all circumstances. Specifying ρ_A, when available[2], gives extra information to the math solver that can boost key rate. Furthermore, the Frank Wolfe solver allows you to specify Block diagonal structure present in Alice and Bob's measurements. This can drastically speed up the math solver and provide a more stable result, especially for flag-state squashing maps.

[2] Such as in prepare and measure protocols.

3.2.6 Channel Modules

Listing 3.4: Channel Module Function Signature

```
function [newParams, modParser]...
  = NAMEChannelFunc(params,options,debugInfo)
% detailed description of module
arguments
    params (1,1) struct
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end
%code
end
```

The main purpose of the channel model is to simulate the quantum channel linking Alice and Bob, and the quantum sources and detector set ups they use. The channel is thus responsible for producing joint/conditional measurement expectations for each operator[3]. To aid in simulation, collections of useful channel functions are found in the classes `Rotations`, `Qudit`, and `Coherent`. For example, these function can help you simulate polarization rotations, loss, and depolarization for both qudit and coherent states. If the module requests the observables, then please ensure the expectation values are formatted with the same shape. Otherwise, please use the convention of Alice is indexed down (first index) and Bob across (second index). Furthermore, the convention for decoy analysis is to store the expectations values in a 3D array where the last dimension iterates over the signal intensities.

Alternatively, the user may write a channel model which either interfaces with an external library or loads experimental data and formats it for the key rate module.

Caution 3.2.6: Expectation Values Order

Note the key rate modules can only realistically check that the joint/conditional expectation values have the same shape as the observables. You must ensure that that each expectation value is appropriately matched with its observable.

Caution 3.2.7: Qetlab Bugs

The release of Qetlab found on their site has a major bug when applying channels to subsystems using the `PartialMap` function. Please ensure you are using the latest release from their GitHub page. See Section 1.2 for more details.

3.2.7 Key Rate Modules

Listing 3.5: Key Rate Module Function Signature

```
function [keyRate, modParser]...
```

[3] For finite size protocols this would produce frequency distributions instead.

```
= BasicKeyRateFunc(params,options,mathSolverFunc,debugInfo)
% detailed description of module
arguments
    params (1,1) struct
    options (1,1) struct
    mathSolverFunc (1,1) function_handle
    debugInfo (1,1) DebugInfo
end
%code
end
```

The key rate module contains implements all steps from a security proof to compute a key rate up to (but not including) calculating a lower bound on the relative entropy between the key and Eve. As such, key rate modules demand far more scrutiny while constructing them and it is advisable to cite relevant papers in the module's description. Parameters passed to the key rate module should be thoroughly validated to ensure that they conform to the class of problems the key rate module was designed to solve. The key rate module can be broken down into 3 main tasks:

- Perform any post processing and special proof techniques required, such as coarse graining, applying squashing, determining flag state bounds, and decoy analysis.
- Calculate the cost of error correction. This is usually handled by calling helper functions.
- Directly interacting with the math solver module for calculating relative entropy.

Furthermore, key rate modules are likely the only module a typical user will write or modify that does not return a list of new parameters. In its place, it returns a guaranteed lower bound on the key rate.

Caution 3.2.8: Requires a Security Proof

Just because a key rate module produces a key rate does not mean that it properly follows a security proof. When writing one, verify that the module correctly implements a numerical proof. Where ever possible, link to a paper that contains the full proof in the comments. It is also recommend that you do this for other components used by the key rate module, such as proofs for any used squashing maps, decoy analysis and finite size effects.

Post Processing

Many post processing steps are highly dependent on numerical security proofs. As such, we can only give more general advice and resources for further reading.

- **Course graining**[7]: Due to a combination of constraints on numerical modeling, proof technique and physical setups, the full set of observables and expectation values between Alice and Bob are too difficult to directly work with. One simple tool to help alleviate this problem is *course graining*, which applies a stochastic map to Alice and Bob's observables and expectation values to simplify the statistics. Often times, we use it to reduce a complex protocol to a well studied one at the cost of some key rate. Note, some proof techniques use different levels of course graining for determining error correction and privacy amplification costs.

- **Squashing maps**[7, 8, 9, 10, 11]: Because all practical QKD protocols use photons to transmit signals, we have to deal with a channel that uses an infinite dimensional hilbert space, which is obviously incompatible with any numerical approach to solving convex optimization problems. To get around this, *squashing maps* were created to translate infinite dimensional problems to finite dimensional ones at the cost of a weaker lower bound. Often times, finding a useful squashing map and finite dimensional model can be exceptionally difficult. Therefore, many squashing maps rely on the user first applying a course graining map to simplify the problem.

- **Decoy analysis**[12, 13, 14]: Single photon sources are currently too expensive for large scale use in QKD. Sources utilizing phase-randomized weak coherent pules (WCP) and thermal sources are cheaper and commercially available, but emit a random number of photons with each pulse. If Alice and Bob cannot determine the number of photons sent and received, then an attack can choose to split photons off the signal and measure after announcements. *Decoy analysis* protects from this attack by Alice sending signals with a random intensity selected from a small pool of values. The intensities chosen are appended to the announcements and upper and lower bounds for all expectation values conditioned on signal, intensity choice, and number of photons sent.

Calculating Error Correction Cost

For basic protocols where Alice and Bob's announcements and key map are independent and deterministic, the cost of error correction is simple to calculate. We cover the theory in more detail in Appendix A.6. For the rest of this section the tools we provide assume error correction is proportional to the Shannon limit as in Eq. (A.46). The general error correction cost can be calculated through the helper function `generalECFunc`. It calculates the total cost in bits and requires 3 arguments:

1. A double `fEC` with $f_{\text{EC}} \geq 1$. This represents the efficiency of error correction. If the Shannon limit has a cost of m bits, then we need fm bits instead.

2. A multidimensional array `gains` with elements $\{p_i\}|_{i=1}^{n}$ with $0 \leq p_i$ and $\sum_i p_i \leq 1$. These represent the probability of each public announcement combination that is accepted.

3. A multidimensional cell array (with the same shape as `gains`) `jointKey`, where each element is the joint distribution of the key dit (first index/down) and Bob's measurement outcome (across/second index) conditioned on the corresponding kept announcement.

In the following example we show a simple example of a joint distribution between Alice and Bob. We then condition on the desired announcement pairs (both measure in the Z basis and both measure in the X basis) and apply the key map to Alice's side. We then format the input and calculate the error correction cost.

Listing 3.6: Example using generalECFunc

```
%% In the channel module

% some initial distribution
simpleStats = rand(4,4);
simpleStats = simpleStats/sum(simpleStats,"all") % sums to 1
```

```
% simpleStats =
%
%     0.0435    0.0676    0.0700    0.0676
%     0.0944    0.0037    0.0781    0.0176
%     0.0817    0.0875    0.0766    0.0728
%     0.0989    0.0963    0.0404    0.0033

%% In the key rate module

% key map  Alice uses H,D -> 0; V,A -> 1
% announcements Alice and Bob both use H,V -> Z; D,A -> X
% only keep announcement pairs (Z,Z) and (X,X)

% post process the simple stats into the the joint statistics between the
% key and Bob conditioned on each announcement.

statsZZ = simpleStats(1:2,1:2);
gainZZ = sum(statsZZ,"all");
statsZZ = statsZZ/gainZZ; % normalize for conditional p(r,y|C=Z)

statsXX = simpleStats(3:4,3:4);
gainXX = sum(statsXX,"all");
statsXX = statsXX/gainXX; % normalize for conditional p(r,y|C=X)

% package for EC cost

givenECEfficiency = 1; % <- from the preset
gains =[gainZZ,gainXX];
stats = {statsZZ,statsXX};

ECCost = generalECFunc(gains,stats,givenECEfficiency)
% ECCost =
%
%     0.2733
```

If Alice and Bob's announcements are independent of each other, and if the post processing is deterministic, then we can use the `errorCorrectionCost` function to automate this process. To use this function, we first need a lists of the announcements for the POVM elements (both Alice's and Bob's). For example, the BB84 protocol uses basis announcements. If Alice and Bob's POVMs are order the outcomes as H, V, D, A then we add the following to our description.

```
% POVMs ordered H,V,D,A
newParams.announcementsA = ["Z","Z","X","X"];
newParams.announcementsB = ["Z","Z","X","X"];
```

Here, we used the strings `"Z"` and `"X"`, but any string, or data type convertible to strings works. Then, for each announcement pair we keep, we add a `KeyMapElement` to contain the key assignment scheme. In this example, we accept matching basis choices only.

```
% Both accepted pairs use the same key map H,D -> 0 and V,A -> 1
newParams.keyMap = [KeyMapElement("Z","Z",[0,1,0,1]),...
    KeyMapElement("X","X",[0,1,0,1])];
```

Just like before, we can use any strings or data types convertible to strings (like how `[0,1,0,1]` is converted to `["0","1","0","1"]`). The only restrictions are that the announcement pairs should be a subset of the Cartesian product of Alice and Bob's announcements, and no announcement pair can have multiple associated key map elements.

After adding these new parameters to your key rate module's input list, you can calculate the error correction cost with,

```
deltaLeak = errorCorrectionCost(params.announcementsA,params.announcementsB,...
    params.expectationsJoint,params.keyMap,params.fEC);
```

Altogether, we get:

Listing 3.7: Example using `errorCorrectionCost`

```
%% In the channel module

% some initial distribution
simpleStats = rand(4,4);
simpleStats = simpleStats/sum(simpleStats,"all") % sums to 1
% simpleStats =
%
%     0.0435    0.0676    0.0700    0.0676
%     0.0944    0.0037    0.0781    0.0176
%     0.0817    0.0875    0.0766    0.0728
%     0.0989    0.0963    0.0404    0.0033

%% In the description module

% announcements Alice and Bob both use H,V -> Z; D,A -> X
announcementsA = ["Z","Z","X","X"];
announcementsB = ["Z","Z","X","X"];

% key map  Alice uses H,D -> 0; V,A -> 1
% only keep announcement pairs (Z,Z) and (X,X)
keyMap = [KeyMapElement("Z","Z",[1,2,1,2]),...
    KeyMapElement("X","X",[1,2,1,2])];


%% In the key rate module

```

```
givenECEfficiency = 1; % <- from the preset
ECCost = errorCorrectionCost(announcementsA,announcementsB,...
    simpleStats,keyMap,givenECEfficiency)
% ECCost =
%
%     0.2733
```

Interacting with the Math Solver

Unlike other modules, key rate modules are given access to the math solver module via the function handle `mathSolverFunc`. This way, more complex protocols can call the math solver module multiple times and perform any additional post processing on the relative entropy returned. Key rate modules are also tasked with constructing custom parameter struct inputs for each call of the math solver. Details of the individual parameters can be found later in Section 3.2.8. For example, we start by adding a new leaf to our debug information to store the math solver's debug info.

```
% Add a new debug leaf to your debugInfo.
debugMathSolver = debugInfo.addLeaves("mathSolver");

% Use the function handle mathSolverFunc to call your math solver. The
% returned relative entropy is measured in bits.
relEnt = mathSolverFunc(mathSolverInput,debugMathSolver);
```

Many protocols may need to rescale the relative entropy afterwards or even calculate the relative entropy of multiple set ups. If multiple calls to the math solver module are required, remember to give each call a separate and unique debug information leaf. More information may be found in Section 3.2.4. Afterwards, the key rate is assembled from the previous components and the key rate is printed to the console if the verbose level is high enough.

```
%store the key rate (even if negative)
keyRate = relEnt-deltaLeak;

if options.verboseLevel>=1
    %ensure that we cut off at 0 when we display this for the user.
    fprintf("Key rate: %e\n",max(keyRate,0));
end
```

Caution 3.2.9: Do Not Cut the Key Rate Off At 0

Between the error correction cost and numerical methods to lower bound the relative entropy, the reported key rate could be less than 0. We strongly recommend that you *do not* replace the key rate with 0. Many types of unstructured global optimization routines fail in flat regions and this could easily break them. A print statement to the console of 0 should be fine though.

3.2.8 Math Solver Modules

Listing 3.8: Math Solver Module Function Signature

```
function [relEntLowerBound, modParser]...
  = NAMEMathSolverFunc(params,options,debugInfo)
% detailed description of module
arguments
    params (1,1) struct
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end
%code
end
```

A math solver module's only goal is to provide a safe *lower bound* on the relative entropy of

$$\min_{\rho_{AB}\in S} \mathrm{D}\left(\mathcal{G}(\rho)\,\|\,\mathcal{Z}\circ\mathcal{G}(\rho)\right) \tag{3.1}$$

which is part of the key rate formula from Eq. (A.1) in Appendix A. The `FW2StepSolver` module that ships with the software uses numerical methods derived from [1]. Further details can be found in Appendices A and B. For now we, will focus only on describing the input and output format.

Easiest to describe, the output is the relative entropy in bits between the key register and a third party with quantum memory and access to all public classical communication. Math solver modules receive a separate struct of parameters crafted for it by the key rate module. These parameters include:

- `krausOps`: A 1D cell array of matrices that represent the $\mathcal{G}$ map Eq. (A.20). They form a completely positive, trace non-increasing map.
- `keyProj`: a 1D cell array of orthogonal projection operators that act on the key register (and are identity on all other registers). They must form a POVM.
- `equalityConstraints`, `inequalityConstraints`, `vectorOneNormConstraints`, and `matrixOneNormConstraints`: These store 1D object arrays of types `EqualityConstraint`, `InequalityConstraint`, `VectorOneNormConstraint`, and `MatrixOneNormConstraint` respectively. These form the bulk of the constraint set S. Although optional, at least one is typically used.
- `rhoA`: An optional parameter for prepare and measure protocols which fixes the value of Alice's marginal distribution $\rho_A := \mathrm{Tr}_{A'}[\rho_{AA'}]$.
- `BlockDimsA` and `BlockDimsB`: These optional constraints help signal to the solver that ρ_{AB} has an exploitable block diagonal structure, which can drastically improve stability and running time.

With the exception of block dimensions, all math solvers should accept the above parameters, though individual math solver modules may request more required or optional parameters.

Caution 3.2.10: Warn Users of Unused Parameters

To ensure that users are made explicitly aware when math solvers ignore certain input parameters (which could include unsupported constraint types), all math solver modules should enable their input parsers to warn about unused parameters. This is done by setting the name-value pair input `warnUnusedParams` to true when calling the module parser's `parse` method. For example:

```matlab
% Replace:
modParser.parse(params);
% With:
modParser.parse(params,"warnUnusedParams",true);
```

The Four Major Constraint Classes

In this small section we give a little more description to each of the four major constraint classes, and an example math solver input struct. The four major constraint classes represent the following types of constraints:

- `EqualityConstraint`:
  - $\mathrm{tr}[\texttt{operator}\rho_{AB}] = \texttt{scalar}$
  - `operator`, hermitian matrix the same size as ρ_{AB}.
  - `scalar`, real value.
- `InequalityConstraint`:
  - $\texttt{lowerBound} \leq \mathrm{tr}[\texttt{operator}\rho_{AB}] \leq \texttt{upperBound}$
  - `operator`, hermitian matrix the same size as ρ_{AB}.
  - `lowerBound`, real value lower bound on inner product.
  - `upperBound`, real valued upper bound on inner product.

  You can set either the upper or lower bound to ∞ and $-\infty$ respectively to ignore that side of the bound.
- `VectorOneNormConstraint`:
  - $\left\|\sum_{i=1}^{n} \mathrm{tr}[\texttt{operators}_i\rho_{AB}] \left|i\right\rangle - \left|\texttt{vector}\right\rangle\right\|_1 \leq \texttt{scalar}$
  - $\left\|\cdot\right\|_1$ is the l_1 norm.
  - `operators`, cell array of n hermitian matrices each the same size as ρ_{AB}.
  - `vector` $n \times 1$ vector of real numbers.
  - `scalar`, non-negative real valued scalar.
- `MatrixOneNormConstraint`:
  - $\left\|\texttt{map}_{AB\to C}(\rho_{AB}) - \texttt{operator}\right\|_1 \leq \texttt{scalar}$.
  - $\left\|\cdot\right\|_1$ is the trace norm.

  - `map`$_{AB\to C}$, function that represents a hermitian preserving super operator from systems AB to C. It is stored as a Choi matrix (called `ChoiMatrix`) with systems ordered ABC.
  - `operator`, a hermitian matrix in system C.
  - `scalar`, non-negative real valued scalar.

Finally, here is a basic example of a list of parameters generated by a key rate function for a math solver.

```
mathSolverInput = struct(); % Make a struct to hold the input.

% Assign the G map's Kraus operators and Z maps projection operators.
mathSolverInput.krausOps = params.krausOps; % cell array
mathSolverInput.keyProj = params.keyProj; % cell array

% Our protocol has equality constraints from observables and expectations.
% Assume we have joint observables and expectations defined earlier and
% stored in the variables obsevablesJoint (cell array) and expectationsJoint
% (array).

numObs = numel(observablesJoint);
% Initialize numObs amount of EqualityConstraint objects
equalityConstraints(numObs) = EqualityConstraint();

for index = 1:numObs
    % add Equality new equality constraints with operators from
    % observablesJoint, and scalars from ExpectationsJoint
    equalityConsraints(index) = EqualityConstraint(...
        observablesJoint{index}, expectationsJoint(index));
end

mathSolverInput.equalityConstraints = equalityConstraints;
```

Optional Math Solver Module Constraints

The following constraints are also available in the Frank-Wolfe 2 step solver, but may not be implemented in all others (though we strongly encourage it).

- `rhoA` constraint:
  - $\mathrm{Tr}_B[\rho_{AB}] =$ `rhoA`
  - `rhoA`, density matrix of size $\dim(A)$.

  For prepare and measure protocols, the source replacement scheme requires (See Appendix A.3.2) requires an additional constraint to prevent the math solver from optimizing Alice's system. This is equivalent to saying that Eve cannot modify Alice's system. For stability, the Frank-Wolfe 2 step solver also adds the explicit constraint $\mathrm{Tr}[\rho_{AB}] = 1$. We assume that ρ_{AB} orders its systems $A \otimes B$.

- `BlockDimsA` and `BlockDimsB`:
  - `BlockDimsA` and `BlockDimsB` are vectors of positive integers that satisfy $\sum_i \texttt{BlockDimsA}_i = \dim(A)$, and $\sum_i \texttt{BlockDimsB}_i = \dim(B)$ respectively.

  These constraints are used to reduce the total number of free parameters needed by the math solver module. For the Frank-Wolfe 2 step solver module, this provides an overall improvement to speed and stability in. Each element contains the size of Alice/Bob's square blocks, ordered from top left to bottom right. The Frank-Wolfe 2 step solver module exploits further block diagonal structure through permutation tricks between Alice and Bob's blocks. We assume that ρ_{AB} orders its systems $A \otimes B$.

Caution 3.2.11: Carefully Check the Block Diagonal Structure

Note that the `FW2StepSolver` math solver module currently does not check if the other parameters have the given block diagonal structure. Please manually check `krausOps` for the $\mathcal{G}$ map, `rhoA` for Alice's marginal, and the various operators and maps for the four major constraint types.

3.2.9 Optimizer Modules

Listing 3.9: Optimizer Module Function Signature

```
function [optimalKeyRate, optimalParams]...
= NAMEOptimizerFunc(optimizeParams,wrappedProtocol,options,debugInfo)
% detailed description of module
arguments
    optimizeParams (1,1) struct
    wrappedProtocol (1,1) function_handle
    options (1,1) struct
    debugInfo (1,1) DebugInfo
end
%code
end
```

Optimizer modules allow the user to perform local or global unstructured parameter optimization for bounded real valued parameters. For example, a weak coherent pulse based protocol will need to alter its choice of signal and decoy intensities for different expected noise or loss parameters. For each iteration, the Optimizer module will determine optimal choices of marked parameters to increase the key rate. See Section 3.1.1 for how to mark parameters for optimization in the preset.

Optimizer modules have the most unique function signature and parsing process. Like all modules, they accept optional parameters and DebugInfo objects, and use those as usual. All other aspects though are significantly more unique. Parameters to optimize over come packaged in `optimizePaarams`, a struct of structs. Each sub structure uses its parameter's name and contains 3 values:

- `lowerBound` lower bound on the optimization parameter.

- `upperBound` upper bound on the optimization parameter.
- `initVal` initial value to start the optimization routine at.

All 3 are real valued scalars and `lowerBound` $\leq$ `initVal` $\leq$ `upperBound`. Because each individual parameter must be validated, the `optimizerValidateProperties` function will handle parsing for the user.

```matlab
%% param parser

modParser = makeOptimizerParamParser(mfilename);
[optimizeParams,~] = optimizerValidateProperties(optimizeParams,modParser);
```

The optimization module also receives the function handle `wrappedProtocol` to evaluate the protocol's key rate. It takes in a single struct with name value pairs of parameter name and value. For example, if we are optimizing over parameters `param1` and `param2` with initial values of 0.5 and 1 respectively, then the first call to the wrapped key rate function may look like

```matlab
currentParams = struct();
currentParams.param1 = 0.5;
currentParams.param2 = 1;

initialKeyRate = wrappedProtocol(currentParams);
```

After determining the optimal parameter choices, the optimizer module returns the optimal key rate and the optimal parameter choices formatted in the same way as above. The software then runs the protocol once more using the optimal parmeters.

When using an optimizer module, global options and individual module options can be overwritten. This can be useful for reducing the verbose level and to tweak the number of steps and tolerances used in expensive calculations. See Section 3.2.3 for order options are overwritten. Here is a quick example of how to set the optimizer module's options, as well as overwriting the global options.

```matlab
% Optimizer module's options
optimizerOptions = struct();
optimizerOptions.maxIterations= 10;

% Global options override for wrapped protocol
globalOptionsOverride = struct();
globalOptionsOverride.verboseLevel = 0;

optimizerMod = QKDOptimizerModule(@coordinateDescentFunc,optimizerOptions,
    globalOptionsOverride);
qkdInput.setOptimizerModule(optimizerMod);
```

Chapter 4

Miscellaneous

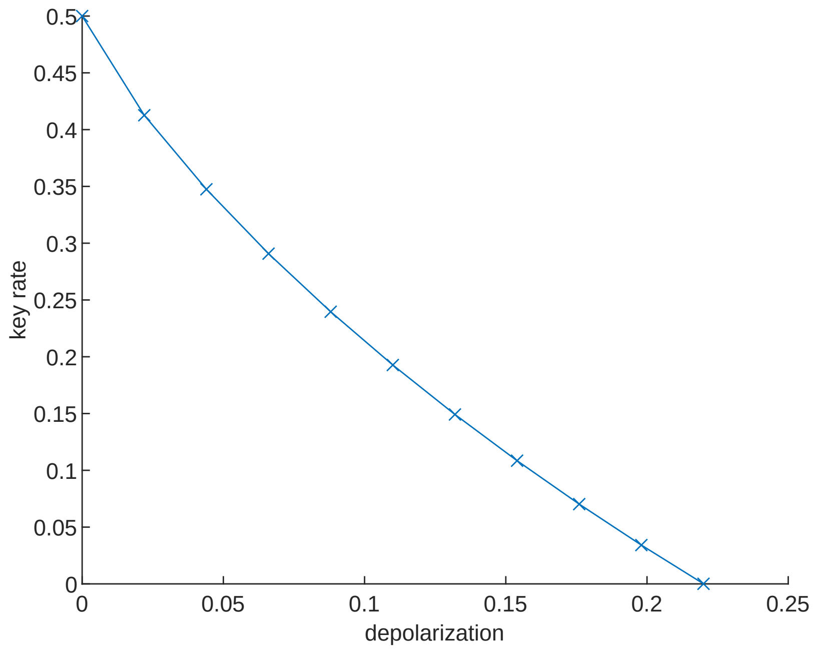
4.1 Plotting Functions

The QKDPlot static class provides a few useful methods for parsing result structure arrays returned from MainIteration and transform them into plots. There are 3 available functions, simple1DPlot, plotParametersFromFiles, and plotParameters. When using any of the three plotting methods, if the parameter selected to plot is not list who's elements are real scalar values, then the plotting functions will fall back to plotting the indices of the elements instead.

4.1.1 simple1DPlot

The simple1DPlot provides a simple method to plot key rates after running MainIteration. Provided there is exactly 1 scan parameter, the function will use the provided QKDSolverInput to determine the scan parameter and generate a plot of the scan parameter versus the key rate without any further input from the user. For example, the BasicBB84Alice2DPreset has only one scan parameter, depolarization. So we can use the simple1DPlot method on it to produce a plot of depolarization vs key rate.

```

% load preset and get key rate results
qkdInput = BasicBB84Alice2DPreset();
results = MainIteration(qkdInput);

% plot
QKDPlot.simple1DPlot(qkdInput,results);
```



4.1.2 plotParametersFromFiles and plotParameters

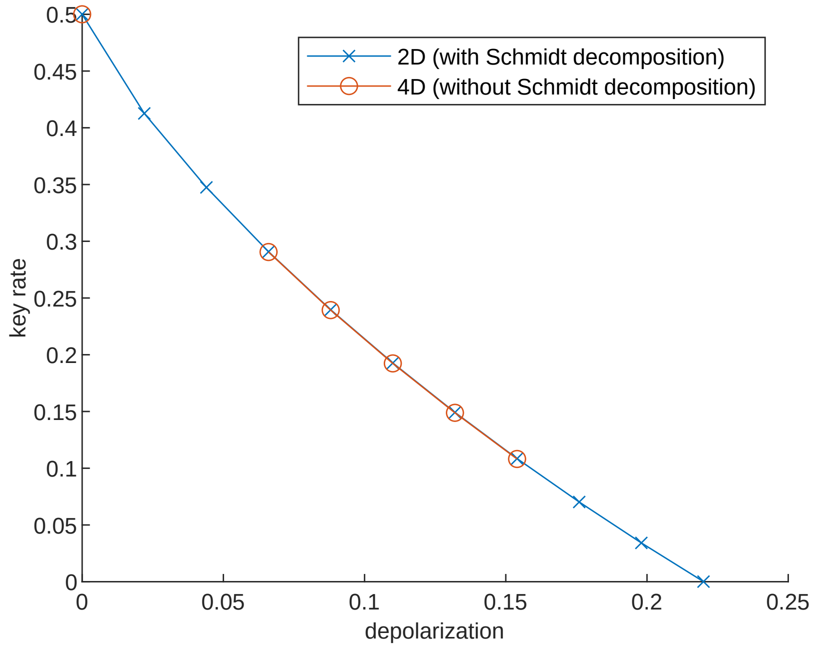
The methods `plotParametersFromFiles` and `plotParameters` allow for greater flexibility when plotting. Both allow for selecting which parameters should be plotted, and on what axis.Additionally, They can allow for multiple sets of results to be plotted on the same graph. The `plotParameters` method has 3 required inputs.

- `resultsSets`: A cell array of struct arrays. Each struct array represents the output of one call to `MainIteration`. This contains the results we want to plot from each set.
- `xParamName`: The name of the parameter you want to plot on the x-axis.
- `yParamName`: The name of the parameter you want to plot on the y-axis.

For `xParamName` and `yParamName`, we can use the special string “_keyRate_” to plot the key rate along that axis. Furthermore, when plotting multiple data sets on the same plot, each data set must have the selected parameters in them. In the following example we use the `plotParameters` to visually show how using the Schmidt decomposition can increase solver stability when reducing dimensions.

```
% load presets and get key rate results
qkdInput2D = BasicBB84Alice2DPreset();
results2D = MainIteration(qkdInput2D);

qkdInput4D = BasicBB84Alice4DPreset();
results4D = MainIteration(qkdInput4D);

```

```
%% plot
resultsSets = {results2D,results4D};
legendNames = ["2D (with Schmidt decomposition)",...
    "4D (without Schmidt decomposition)"];

QKDPlot.plotParameters(resultsSets,"depolarization","_keyRate_",...
    "legendNames",legendNames)
```



Caution 4.1.1: Multiple data sets and index fall back

When plotting multiple data sets with `plotParametersFromFiles` and `plotParameters`, if the parameter cannot be converted to a list who's elements are real valued scalar, then the index of the elements will be plotted instead. When this occurs, each data set must have the same number of points to plot.

The `plotParametersFromFiles` works identically to `plotParameters` except it takes in an array of file names/-path strings. These files must contain a struct array variable called `results` which is the output of one call to `MainIteration`. The following example is an equivalent method to generating the same plot using `plotParametersFromFiles`.

```
% load presets and get key rate results
qkdInput = BasicBB84Alice2DPreset();
results = MainIteration(qkdInput);

save("BasicBB84Alice2DResults.mat","results","qkdInput");

```

```
qkdInput = BasicBB84Alice4DPreset();
results = MainIteration(qkdInput);

save("BasicBB84Alice4DResults.mat","results","qkdInput");

%% plot
fileNames = ["BasicBB84Alice2DResults.mat","BasicBB84Alice4DResults.mat"];
legendNames = ["2D (with Schmidt decomposition)",...
    "4D (without Schmidt decomposition)"];

QKDPlot.plotParametersFromFiles(fileNames,"depolarization","_keyRate_",...
    "legendNames",legendNames)
```

4.1.3 Common Plot Options

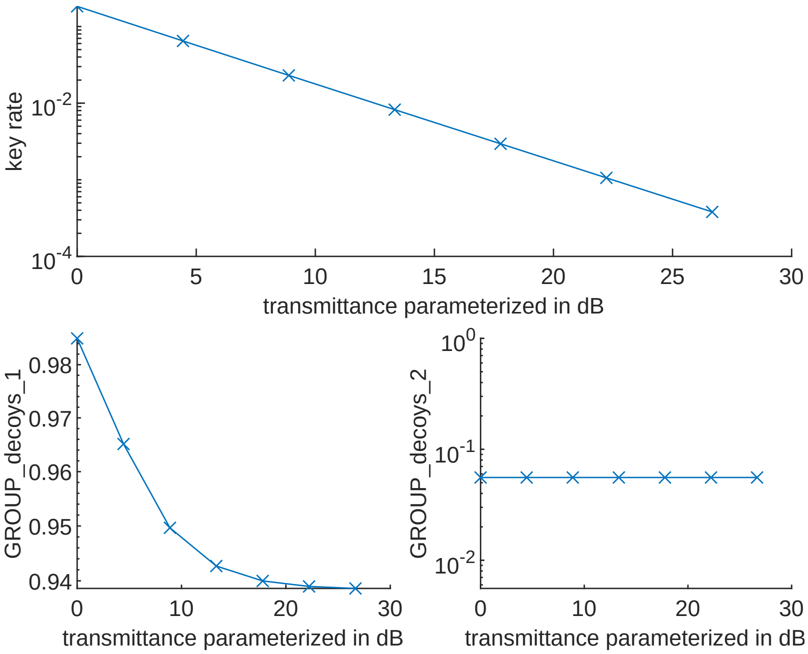
Although each plotting function has slightly different ways of loading data, they share many common options. Each support the following name-value pair arguments:

- `addLegend`: Logical value to toggle the legend on (default) or off. (Not supported by `simple1DPlot` which has no legend).
- `legendNames`: String array containing the names to use for each data set plotted on the same axis. (Not supported by `simple1DPlot`). By default, the names will be the "data set x" where x is given by the order results where given to the plotting function.
- `XScaleStyle` and `yScaleStyle`: Strings of either "linear" (default), "log", or "dB". These control how the axis will scale. Linear and log are linear and logarithmic scaling for the axis, respectively. The dB option instead applies the transformation `@(x)-10*log10(x)` to that axis. The dB option is only supported for real, scalar valued data points.
- `markerLineStyles`: A string array of Matlab marker line styles for the plotting function to cycle between. The styles follow Matlab's plotting function format. For example, "x-" uses solid lines and x to mark the points. If there are more data sets than marker line styles, the styles will be looped through again from the beginning.
- `figAxis`: A `matlab.graphics.axis.Axes` object to plot on. If none is supplied, then a new figure is created to plot on instead. Useful for adding QKD plots to subplots.

In the following example, we optimize over the signal intensity (`GROUP_decoys_1`) and the first intensity used purely for decoy analysis (`GROUP_decoys_2`), then plot all three in the same on different tiles of the same figure.

```
% load presets and get key rate results
qkdInput = BasicBB84WCPDecoyPreset();

optimizeBounds1 = struct("lowerBound",0,"initVal",0.45,"upperBound",1);
qkdInput.addOptimizeParameter("GROUP_decoys_1", optimizeBounds1);

```

```
optimizeBounds2 = struct("lowerBound",0,"initVal",0.1,"upperBound",1);
qkdInput.addOptimizeParameter("GROUP_decoys_2", optimizeBounds2);

results = MainIteration(qkdInput);

results = results(1:end-3); %area where it converged well for sample plot


%% plot
tiles = tiledlayout(2,2,"TileSpacing","tight","Padding","tight");

axis1 = nexttile(tiles,[1,2]);
QKDPlot.plotParameters({results},"transmittance","_keyRate_",...
    "addLegend",false,"xScaleStyle","dB","yScaleStyle","log",...
    "figAxis",axis1);

axis2 = nexttile(tiles);
QKDPlot.plotParameters({results},"transmittance","GROUP_decoys_1",...
    "addLegend",false,"xScaleStyle","dB","yScaleStyle","log",...
    "figAxis",axis2);

axis3 = nexttile(tiles);
QKDPlot.plotParameters({results},"transmittance","GROUP_decoys_2",...
    "addLegend",false,"xScaleStyle","dB","yScaleStyle","log",...
    "figAxis",axis3);
```



4.2 Diagnosing Common Problems

Expand Q and A section from feedback.

- **Console has "linsysolve: solution contains NaN or inf" printed in random places.**

  This is a known bug with SDPT3, the default solver we use with CVX. There is unfortunetly nothing we can do about it.

References

[1] Adam Winick, Norbert Lütkenhaus, and Patrick J. Coles. “Reliable Numerical Key Rates for Quantum Key Distribution”. In: *Quantum* 2 (July 2018), p. 77. ISSN: 2521-327X. DOI: 10.22331/q-2018-07-26-77. arXiv: 1710.05511 [quant-ph].

[2] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th anniversary ed. Cambridge ; New York: Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.

[3] John Watrous. *The Theory of Quantum Information*. 1st ed. Cambridge University Press, Apr. 2018. ISBN: 978-1-316-84814-2 978-1-107-18056-7. DOI: 10.1017/9781316848142.

[4] Mark Wilde. *Quantum Information Theory*. Second edition. Cambridge, UK ; New York: Cambridge University Press, 2017. ISBN: 978-1-107-17616-4.

[5] Ramona Wolf. *Quantum Key Distribution: An Introduction with Exercises*. Vol. 988. Lecture Notes in Physics. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-73990-4 978-3-030-73991-1. DOI: 10.1007/978-3-030-73991-1.

[6] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge, UK ; New York: Cambridge University Press, 2004. ISBN: 978-0-521-83378-3.

[7] O. Gittsovich et al. “Squashing Model for Detectors and Applications to Quantum-Key-Distribution Protocols”. In: *Physical Review A* 89.1 (Jan. 2014), p. 012325. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.89.012325.

[8] Normand J. Beaudry, Tobias Moroder, and Norbert Lütkenhaus. “Squashing Models for Optical Measurements in Quantum Communication”. In: *Physical Review Letters* 101.9 (Aug. 2008), p. 093601. DOI: 10.1103/PhysRevLett.101.093601.

[9] Chi-Hang Fred Fung, H. F. Chau, and Hoi-Kwong Lo. “Universal squash model for optical communications using linear optics and threshold detectors”. In: *Phys. Rev. A* 84 (2 2011), p. 020303. DOI: 10.1103/PhysRevA.84.020303.

[10] Yanbao Zhang et al. “Security proof of practical quantum key distribution with detection-efficiency mismatch”. In: *Phys. Rev. Res.* 3 (1 2021), p. 013076. DOI: 10.1103/PhysRevResearch.3.013076.

[11] Twesh Upadhyaya et al. “Dimension Reduction in Quantum Key Distribution for Continuous- and Discrete-Variable Protocols”. In: *PRX Quantum* 2 (2 2021), p. 020325. DOI: 10.1103/PRXQuantum.2.020325.

[12] Xiongfeng Ma et al. “Practical decoy state for quantum key distribution”. In: *Phys. Rev. A* 72 (1 2005), p. 012326. DOI: 10.1103/PhysRevA.72.012326.

[13] Nicky Kai Hong Li and Norbert Lütkenhaus. “Improving key rates of the unbalanced phase-encoded BB84 protocol using the flag-state squashing model”. In: *Phys. Rev. Res.* 2 (4 2020), p. 043172. DOI: 10.1103/PhysRevResearch.2.043172.

[14] Wenyuan Wang and Norbert Lütkenhaus. “Numerical security proof for the decoy-state BB84 protocol and measurement-device-independent quantum key distribution resistant against large basis misalignment”. In: *Phys. Rev. Res.* 4 (4 2022), p. 043097. DOI: 10.1103/PhysRevResearch.4.043097.

[15] Patrick J. Coles. “Unification of Different Views of Decoherence and Discord”. In: *Phys. Rev. A* 85.4 (Apr. 2012), p. 042103. DOI: 10.1103/PhysRevA.85.042103.

[16] Jie Lin, Twesh Upadhyaya, and Norbert Lütkenhaus. “Asymptotic Security Analysis of Discrete-Modulated Continuous-Variable Quantum Key Distribution”. In: *Physical Review X* 9.4 (Dec. 2019), p. 041064. ISSN: 2160-3308. DOI: 10.1103/PhysRevX.9.041064. arXiv: 1905.10896 [quant-ph].

[17] Charles H. Bennett, Gilles Brassard, and N. David Mermin. “Quantum Cryptography without Bell’s Theorem”. In: *Physical Review Letters* 68.5 (Feb. 1992), pp. 557–559. ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.68.557.

[18] Agnes Ferenczi and Norbert Lütkenhaus. “Symmetries in Quantum Key Distribution and the Connection between Optimal Attacks and Optimal Cloning”. In: *Physical Review A* 85.5 (May 2012), p. 052310. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.85.052310. arXiv: 1112.3396 [quant-ph].

[19] Ian George, Jie Lin, and Norbert Lütkenhaus. “Numerical Calculations of the Finite Key Rate for General Quantum Key Distribution Protocols”. In: *Physical Review Research* 3.1 (Mar. 2021), p. 013274. ISSN: 2643-1564. DOI: 10.1103/PhysRevResearch.3.013274.

[20] Alexander Barvinok. *A Course in Convexity.* Vol. 54. Graduate Studies in Mathematics. Providence, RI: American Mathematical Society, Nov. 2002. ISBN: 978-0-8218-2968-4.

[21] Marguerite Frank and Philip Wolfe. “An algorithm for quadratic programming”. In: *Naval Research Logistics Quarterly* 3.1-2 (1956), pp. 95–110. DOI: https://doi.org/10.1002/nav.3800030109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800030109.

[22] Twesh Upadhyaya. “Tools for the Security Analysis of Quantum Key Distribution in Infinite Dimensions”. MA thesis. University of Waterloo, Aug. 2021.

Appendices

Appendix A

d-Dimensional Protocol Theory

In this appendix, we will discuss a generic d-dimensional QKD-protocol and how to incorporate it into our software to calculate its asymptotic key rate. In the process, the BB84 protocol is considered as a running example. We will derive all the necessary quantities needed to set up our software. The appropriate modules in which these are used, include the description and the channel model.

The structure of this guide is as follows. First in Appendix A.1, we will introduce a generic entanglement based protocol and the necessary notations. Furthermore, we present how prepare-and-measure protocols are related to it. Next in Appendix A.2, we describe the process of the key rate calculation in general using the relative entropy formulation.

Following from there, in Appendix A.3 we focus on Alice's and Bob's measurements and announcements. Then in Appendix A.4, we construct the so-called $\mathcal{G}$ and $\mathcal{Z}$-maps needed for the key rate calculation using the relative entropy.

In Appendix A.5 we show how we use simulated data as a channel model and in Appendix A.6 we present the steps to determine the error correction cost. Finally, in Appendix A.7 we present techniques to reduce the dimensions for better numerical stability.

A.1 Generic QKD Protocol

A.1.1 Entanglement Based Protocol

1. **State preparation and transmission** Eve prepares a state $\rho_{A^N B^N E}$, keeps her part E and sends the remaining parts A_i and B_i in round i via a quantum channel to Alice and Bob, respectively.
2. **Measurements** In each round Alice and Bob measure their received states with one of their POVMs $\{M_k^A\}_{k=1\ldots d_A}$, $\{M_k^B\}_{k=1\ldots d_B}$, and store their results in registers X_i and Y_i with alphabets $\mathcal{X}$ and $\mathcal{Y}$.
3. **Public announcement and sifting** Alice and Bob have a public discussion [1] and announce some of their data publicly. They store their public data in $C = C_A C_B$, where C_A and C_B correspond to Alice's and Bob's

[1] This could be multi-round communication.

announcement registers, respectively. Those registers have alphabets $\mathcal{C}_A$ and $\mathcal{C}_B$. Then, Alice and Bob sift their data based on their public communication.

4. **Acceptance test (parameter estimation)** Alice and Bob perform statistical tests on a randomly chosen subset of the data, based on their public announcements. Depending on their results Alice and Bob continue or abort the protocol.

5. **Key map** Alice (or Bob) performs the key map $g : \mathcal{XC} \to \mathcal{R}$, where R is the key register with alphabet $\mathcal{R}$.

6. **Error correction** Using the authenticated channel, Alice and Bob communicate to reach agreeing keys for both parties. In the process, they reveal an amount of δ_{leak} per signal sent.

7. **Privacy Amplification** Alice and Bob randomly choose a universal hash function and apply it to their raw key. After this step, the key is shortened such that the key has a secure length l.

A.1.2 Prepare-and-Measure Protocols

In prepare-and-measure protocols only steps 1. and 2. change. Alice prepares one of d_A states and sends it to Bob through an insecure quantum channel, who then measures it. Now, Alice records her state choice in X. A prepare-and-measure protocol can be recast as an entanglement based protocol. This is done by our software and is presented in Appendix A.3.2.

A.2 Asymptotic Key Rate Formula and To-Do list for programming protocols

Our software uses the methods presented in [1, 15] with the simplifications of [16, App. A] to calculate secret key rates. This guide will mainly focus on the asymptotic regime and mention references to the finite-size where applicable. In the asymptotic regime, of a collective iid attacks (each signal is independently acted on by the same channel), the secret key rate per signal sent is given by

$$R_\infty = \min_{\rho_{AB} \in S \cap \text{Pos}(AB)} \mathrm{D}\left(\mathcal{G}\left(\rho_{AB}\right) \| \mathcal{Z} \circ \mathcal{G}\left(\rho_{AB}\right)\right) - \delta_{\text{leak}}, \tag{A.1}$$

where $\mathrm{D}\left(\cdot \| \cdot\right)$ is the quantum relative entropy. The set S formalizes the constraints on the density matrix ρ_{AB}, given by information such as the observations and $\mathrm{Tr}\left[\rho_{AB}\right] = 1$ (more details in Appendix A.5. The $\mathcal{G}$ and $\mathcal{Z}$ map implement the measurements, announcements, sifting, and the key map. The exact definitions of the set S, and the maps $\mathcal{G}$ and $\mathcal{Z}$ will be explained in the following sections. We will describe how to derive all necessary parts of S, and the maps $\mathcal{G}$, $\mathcal{Z}$ for generic entanglement based and prepare-and-measure protocols.

Independent of the particular protocol, the procedure to implement a protocol in our software is summarized in the following To-Do list.

To-Do List

1. Determine the POVM elements for Bob's measurements and Alice's measurements or state choices.

2. Partition the POVM elements and measurement outcomes in groups with the same public announcement.
3. With the information of steps 1. and 2., construct the Kraus operators of the $\mathcal{G}$ map.
4. Determine the sifting procedure and only keep those Kraus operators which survive sifting.
5. Calculate the Kraus operators of the $\mathcal{Z}$ map.

Finally, one needs to make use of the observations which are either given from simulation or experimental data.

6. Use experimental or simulated observations from a channel model. These observations determine constraints of the form $\mathrm{Tr}[\Gamma_k \rho]$ of the set S.
7. Calculate the gains, i.e. the probability of each announcement that survives sifting, for the error correction cost δ_{leak}.

We will explain each of these steps theoretically and keep the prepare-and-measure qubit BB84 protocol as a rolling example throughout the text.

A.3 POVM elements and announcements

A.3.1 Entanglement Based Protocols

We start with entanglement based protocols. First, assume we have already determined the POVMs of Alice and Bob, given by $\{M_k^A\}_{k=1\dots d_A}$ and $\{M_k^B\}_{k=1\dots d_B}$, respectively. Following our workflow, we then partition Alice's and Bob's POVM elements and measurement results by their public announcements. Starting with our measurement outcomes, we partition Alice's alphabet of measurement outcomes $\mathcal{X}$ into subsets $\mathcal{X}_\alpha$ for $\alpha \in \mathcal{C}_A$ such that

$$\mathcal{X} = \bigcup_{\alpha \in \mathcal{C}_A} \mathcal{X}_\alpha. \tag{A.2}$$

Similarly, one finds for Bob's alphabet of outcomes

$$\mathcal{Y} = \bigcup_{\beta \in \mathcal{C}_B} \mathcal{Y}_\beta. \tag{A.3}$$

Using this we can partition the POVM elements accordingly, such that

$$\{M_k^A\}_{k=1\dots d_A} = \{M_{\alpha,x}^A\}_{\alpha\in\mathcal{C}_A, x\in\mathcal{X}_\alpha}, \tag{A.4}$$
$$\{M_l^B\}_{l=1\dots d_B} = \{M_{\beta,y}^B\}_{\beta\in\mathcal{C}_B, y\in\mathcal{Y}_\beta}. \tag{A.5}$$

The partitioning by public announcements is needed later for constructing the $\mathcal{G}$-map as shown in Appendix A.4. We will use these partitioned POVMs in the following steps.

A.3.2 Prepare and Measure Protocols

In contrast to entanglement based protocols, in prepare-and-measure protocols, Alice (sender) sends signals to Bob (receiver) over a quantum channel. We convert prepare-and-measure protocols to entanglement based protocols because it is easier to analyse in our framework. The source replacement scheme [17, 18] lets us do this conversion.

If Alice sends d_A signal states $\{|s_i\rangle\}_{i=1\ldots d_A}$, where $|s_i\rangle$ is sent with probability p_i, then we can convert this to a an entanglement protocol where Alice prepares the state

$$|\psi\rangle_{AA'} = \sum_{i=1}^{d_A} \sqrt{p_i}\, |i\rangle_A \, |s_i\rangle_{A'} \,. \tag{A.6}$$

Here, she keeps system A and sends system A' to Bob through the channel controlled by Eve. Alice randomly selecting a signal state $|s_i\rangle$ with probability p_i is now equivalent to acting on $|\psi\rangle\langle\psi|_{AA'}$ with the POVM element $M_i^A := |i\rangle\langle i|$ on Alice's system.

For prepare and measure protocols, Alice's choice of the signal state happens under her domain and is inaccessible to Eve, i.e. Eve must act as the identity on Alice's system A. Hence, Eve's action on system AA' can be written as a channel

$$(\mathrm{id}_A \otimes \mathcal{E}) : \mathcal{H}_A \otimes \mathcal{H}_{A'} \to \mathcal{H}_A \otimes \mathcal{H}_B . \tag{A.7}$$

Then the state shared between Alice and Bob is

$$\rho_{AB} = (\mathrm{id}_A \otimes \mathcal{E})\, (|\psi\rangle\langle\psi|_{AA'}) \,. \tag{A.8}$$

Connecting this back to an entanglement based protocol we can equivalently write this attack as Eve distributing the state ρ_{ABE}, where she keeps her system E. This only gives Eve more power, again too much power, since Eve could influence Alice's system A. Thus, we need to enforce that Alice's marginal stays the same, which amounts to

$$\mathrm{Tr}_B\left[\rho_{AB}\right] = \mathrm{Tr}_{A'}\left[|\psi\rangle\langle\psi|_{AA'}\right] , \tag{A.9}$$

This is achieved by specifying Alice's reduced state ρ_A in the description file.

Example A.3.1: BB84 POVM elements

Alice sends the following states through an insecure quantum channel to Bob

$$\{|0\rangle_{A'}, |1\rangle_{A'}, |+\rangle_{A'}, |-\rangle_{A'}\} \,. \tag{A.10}$$

She sends states in the Z-basis ($|0\rangle_{A'}$ and $|1\rangle_{A'}$) with probability p_z and in the X-basis with probability $p_x = 1 - p_z$. Within each basis, Alice chooses with equal probability between the signals.
Using the source replacement scheme we convert this setup to Alice sending a state

$$|\psi\rangle_{AA'} = \sqrt{\frac{p_z}{2}}\, (|00\rangle_{AA'} + |11\rangle_{AA'}) + \sqrt{\frac{1-p_z}{2}}\, (|2+\rangle_{AA'} + |3-\rangle_{AA'}) \,, \tag{A.11}$$

and measuring system A with her POVM elements depending on the state she intended to send. Hence, Alice's POVM elements are

$$M^A = \{|0\rangle\langle 0| \,, |1\rangle\langle 1| \,, |2\rangle\langle 2| \,, |3\rangle\langle 3|\}, \tag{A.12}$$

and her reduced state is

$$\rho_A = \mathrm{Tr}_{AA'}\left[|\psi\rangle\langle\psi|_{AA'}\right]. \tag{A.13}$$

Bob measures his system with the POVM given by

$$M^B = \{p_z\,|0\rangle\langle 0|\,, p_z\,|1\rangle\langle 1|\,, (1-p_z)\,|+\rangle\langle +|\,, (1-p_z)\,|-\rangle\langle -|\} \tag{A.14}$$

Note that the first two POVM elements measure in the Z-basis, while the second two measure in the X-basis. In standard BB84, Alice and Bob announce their bases, hence for instance Alice's partitioned alphabet can be recast as

$$\mathcal{X} = \bigcup_{\alpha\in\{X,Z\}} \mathcal{X}_\alpha, \tag{A.15}$$

where $\mathcal{X}_\alpha = \{0,1\}$ for all α. The same expression holds for Bob's register. Thus, partitioning Alice's and Bob's POVMs by public announcements results in

$$M^A_{Z,0} = |0\rangle\langle 0|\,, \qquad M^A_{X,0} = |2\rangle\langle 2|\,, \tag{A.16}$$

$$M^A_{Z,1} = |1\rangle\langle 1|\,, \qquad M^A_{X,1} = |3\rangle\langle 3|\,, \tag{A.17}$$

and

$$M^B_{Z,0} = p_z\,|0\rangle\langle 0|\,, \qquad M^B_{X,0} = (1-p_z)\,|+\rangle\langle +|\,, \tag{A.18}$$

$$M^B_{Z,1} = p_z\,|1\rangle\langle 1|\,, \qquad M^B_{X,1} = (1-p_z)\,|-\rangle\langle -|\,. \tag{A.19}$$

This is the canonical representation of the POVM elements. In later parts, see Appendix A.7, we reduce Alice's POVM elements to save dimensions.

A.4 $\mathcal{G}$ and $\mathcal{Z}$ map

The steps 2., 3. and 5. of the generic entanglement based protocol described in Appendix A.1 can be combined in the so-called $\mathcal{G}$-map, see [1]. Because of the large overhead of the formulation in [1], we use the simplified version presented in [16]. We define the Kraus operators of the $\mathcal{G}$-map first. They are partitioned by the public announcements of Alice and Bob respectively and the $\mathcal{G}$-map is defined as

$$\mathcal{G}(\rho_{AB}) = \sum_{\alpha\in\mathcal{C}_A,\beta\in\mathcal{C}_B} K_{\alpha,\beta}\rho_{AB}K^\dagger_{\alpha,\beta}. \tag{A.20}$$

Adapting [16, eq. (A9)] for a direct reconciliation protocol, the Kraus operators are given by

$$K_{\alpha,\beta} = \Pi^{\mathrm{sift}}_{RXABC} \sum_{x\in\mathcal{X}_\alpha} |g(x,\alpha,\beta)\rangle_R \otimes |x\rangle_X \otimes \sqrt{M^A_{\alpha,x}} \otimes \sqrt{M^B_\beta} \otimes |\alpha,\beta\rangle_C\,, \tag{A.21}$$

where

$$M^B_\beta = \sum_{y\in\mathcal{Y}_\beta} M^B_{\beta,y}. \tag{A.22}$$

In this formulation,

$$\Pi^{\text{sift}}_{RXABC} = I_{RXAB} \otimes \Pi^{\text{sift}}_{C} \tag{A.23}$$

implements the sifting step, and thus removes all Kraus operators corresponding to sifted announcements. The map $g(x, \alpha, \beta)$ again performs the key map. Technically, one would need to make sure that all $\mathcal{X}_\alpha$ are the same alphabet[1]. In [16] this is done rigorously, but we leave it out for simplicity and convenience in notation.

If the protocol uses reverse reconciliation, the roles of Alice and Bob are reversed. The resulting Kraus operators are

$$K_{\alpha,\beta} = \Pi^{\text{sift}}_{RYABC} \sum_{y\in\mathcal{Y}_\beta} |g(y, \alpha, \beta)\rangle_R \otimes |y\rangle_Y \otimes \sqrt{M^A_\alpha} \otimes \sqrt{M^B_{\beta,y}} \otimes |\alpha, \beta\rangle_C \,, \tag{A.24}$$

where

$$M^A_\alpha = \sum_{x\in\mathcal{X}_\alpha} M^A_{\alpha,x}, \text{ and} \tag{A.25}$$

$$\Pi^{\text{sift}}_{RYABC} = I_{RYAB} \otimes \Pi^{\text{sift}}_{C}. \tag{A.26}$$

This corresponds to the original formulation of [16, eq. (A9)] with the notation adapted for this guide.

In both cases of direct and reverse reconciliation, if the key map simply copies the value of register X (or Y), we can remove the register X (or Y) from the Kraus operators. Although, there are more simplifications which can be done, it is more instructive to show an example at this point and we will continue with further simplifications afterwards.

Example A.4.1: BB84 Kraus operators $\mathcal{G}$ map

In standard BB84, Alice and Bob announce their bases and as presented in example A.3.1 the partitioning results in Alice's and Bob's measurement outcome to be labelled by either 0 or 1 independent of the announcement. Therefore, the key map g can be succinctly written as,

$$g(x, \alpha, \beta) = x, \tag{A.27}$$

effectively copying x to the key register.
As pointed out in Eq. (A.33), the register X can be removed from Eq. (A.21). First, we calculate Bob's summed POVM elements given the particular announcement using his partitioned POVM elements from example A.3.1 and inserting them into Eq. (A.22). We find

$$M^B_Z = \sum_{y\in\mathcal{Y}_Z} M^B_{Z,y} = \sum_{y\in\{0,1\}} M^B_{Z,y} = p_z I_B, \tag{A.28}$$

$$M^B_X = \sum_{y\in\mathcal{Y}_X} M^B_{X,y} = \sum_{y\in\{0,1\}} M^B_{X,y} = (1 - p_z)\, I_B. \tag{A.29}$$

Next, we can insert these results into our general expression for the Kraus operators Eq. (A.21) and use Alice's

[1] One way of doing this is to pick a' such that $\mathcal{X}_{a'}$ is the largest alphabet. Then, relabel elements from the other alphabets with elements from $\mathcal{X}_{a'}$.

partitioned POVM elements from example A.3.1. The Kraus operators of the $\mathcal{G}$-map before sifting are

$$\begin{aligned}
K_{Z,Z} &= \sum_{a\in\{0,1\}} |g(a,Z,Z)\rangle_R \otimes \sqrt{M^A_{Z,a}} \otimes \sqrt{M^B_Z} \otimes |Z,Z\rangle_C \\
&= \sqrt{p_z}\left(|0\rangle_R \otimes |0\rangle\langle 0|_A + |1\rangle_R \otimes |1\rangle\langle 1|_A\right) \otimes I_B \otimes |00\rangle_C \\
K_{X,Z} &= \sum_{a\in\{0,1\}} |g(a,X,Z)\rangle_R \otimes \sqrt{M^A_{X,a}} \otimes \sqrt{M^B_Z} \otimes |X,Z\rangle_C \\
&= \sqrt{p_z}\left(|0\rangle_R \otimes |0\rangle\langle 0|_A + |1\rangle_R \otimes |1\rangle\langle 1|_A\right) \otimes I_B \otimes |10\rangle_C \\
K_{Z,X} &= \sum_{a\in\{0,1\}} |g(a,Z,X)\rangle_R \otimes \sqrt{M^A_{Z,a}} \otimes \sqrt{M^B_X} \otimes |Z,X\rangle_C \\
&= \sqrt{1-p_z}\left(|0\rangle_R \otimes |2\rangle\langle 2|_A + |1\rangle_R \otimes |3\rangle\langle 3|_A\right) \otimes I_B \otimes |01\rangle_C \\
K_{X,X} &= \sum_{a\in\{0,1\}} |g(a,X,X)\rangle_R \otimes \sqrt{M^A_{X,a}} \otimes \sqrt{M^B_X} \otimes |X,X\rangle_C \\
&= \sqrt{1-p_z}\left(|0\rangle_R \otimes |2\rangle\langle 2|_A + |1\rangle_R \otimes |3\rangle\langle 3|_A\right) \otimes I_B \otimes |11\rangle_C \,,
\end{aligned} \tag{A.30}$$

where we used $Z \to 0,\ X \to 1$ in the announcement register C. This is an arbitrary choice and we made this assignment out of convenience.

Next, the sifting is based on the public announcements and a key is only generated if Alice and Bob measure in the same basis, i.e. $\alpha = \beta$. Thus,

$$\begin{aligned}
\Pi^{\text{sift}}_{RABC} &= I_{RAB} \otimes \Pi^{\text{sift}}_C = I_{RAB} \otimes \left(|ZZ\rangle\langle ZZ|_C + |XX\rangle\langle XX|_C\right), \\
&= I_{RAB} \otimes \left(|00\rangle\langle 00|_C + |11\rangle\langle 11|_C\right),
\end{aligned} \tag{A.31}$$

where we have again used the assignment $Z \to 0,\ X \to 1$. Then, the surviving Kraus operators are

$$\begin{aligned}
K_{Z,Z} &= \sqrt{p_z}\left(|0\rangle_R \otimes |0\rangle\langle 0|_A + |1\rangle_R \otimes |1\rangle\langle 1|_A\right) \otimes I_B \otimes |00\rangle_C \,, \\
K_{X,X} &= \sqrt{1-p_z}\left(|0\rangle_R \otimes |2\rangle\langle 2|_A + |1\rangle_R \otimes |3\rangle\langle 3|_A\right) \otimes I_B \otimes |11\rangle_C \,,
\end{aligned} \tag{A.32}$$

which as expected correspond to Alice and Bob measuring in the same basis.

Returning to further simplifications of Eq. (A.21), note that Alice has a register for her measurement outcome x, but Bob does not. The reason for this comes from [16] prior to Eq. (A9)[2], it is reasoned that because only Bob's announcement influences the key, his fine grained information in $|y\rangle_Y$ can be dropped, giving Eq. (A.21). To illustrate the process, in Fig. A.1 we rewrite Bob's measurement process as an isometry V such that $\text{Tr}_B[V\rho_B V^\dagger]$ is equivalent to measuring ρ_B with Bob's POVMs. Then we break V into two steps represented by isometries V_1 and V_2. The first step, V_1, produces his announcement, and the second, V_2, produces the remaining fine grained information. The second isoemtry, V_2 can be dropped by exploiting the relative entropy's invariance under isoemtries Eq. (A.54). For more details please see [16, App. A].

However, one can use a similar argument to exclude the register X (or Y for reverse reconciliation) as well. Then

[2] Their original argument was laid out for a reverse reconciliation protocol while we are discussing direct.

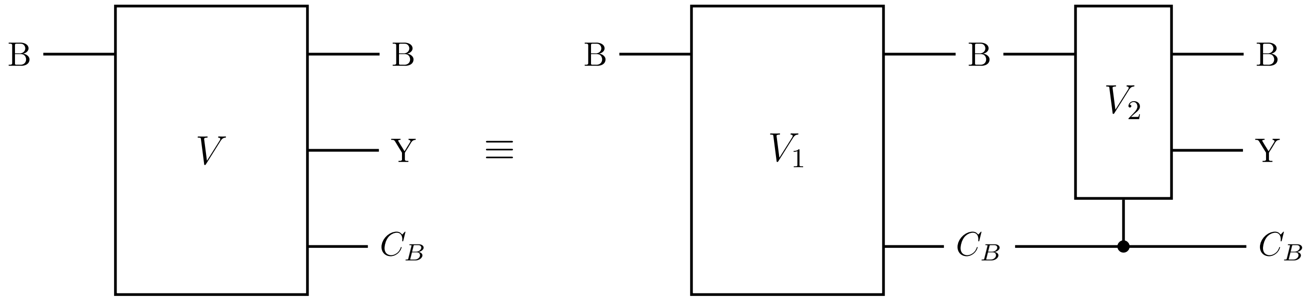


Figure A.1: Equivalence of Bob's measurement isometry as two consecutive measurements also represented as isometries.

the Kraus operators simplify to

$$K_{\alpha,\beta} = \Pi^{\text{sift}}_{RABC} \sum_{r\in\mathcal{R}} |r\rangle_R \otimes \sqrt{M^A_{\alpha,\beta,r}} \otimes \sqrt{M^B_\beta} \otimes |\alpha,\beta\rangle_C \, , \tag{A.33}$$

where now

$$M^A_{\alpha,\beta,r} = \sum_{x\in g^{-1}(\alpha,\beta,r)} M^A_{\alpha,x}. \tag{A.34}$$

The same applies to reverse reconciliation with reversed roles for Alice and Bob[3].

Apart from the $\mathcal{G}$-map, the $\mathcal{Z}$ map is needed which removes all off-diagonal elements in the key register R. It acts as the identity on all registers apart from R, and thus can be written in terms of Kraus operators as

$$\mathcal{Z}\left(\rho_R\right) = \sum_{r\in\mathcal{R}} P_r \rho_R P_r^\dagger, \tag{A.35}$$

where the projectors P_r are

$$P_r = |r\rangle\!\langle r|_R \, . \tag{A.36}$$

Example A.4.2: BB84 Kraus operators $\mathcal{Z}$ map

The $\mathcal{Z}$-map is simply a projection onto our key register and has the following Kraus operators,

$$P_0 = |0\rangle\!\langle 0|_R \otimes I_{ABC}, \quad P_1 = |1\rangle\!\langle 1|_R \otimes I_{ABC}. \tag{A.37}$$

Finally, we have all the tools for the first term, the relative entropy, in the key rate formula Eq. (A.1), and we will focus on the set S in the next Appendix A.5.

[3] Granted enough patience, one can go beyond simple direct and reverse reconciliation and convert a whole LOCC sequence of measurements to a form like $K_\gamma = \Pi^{\text{sift}}_{RABC} \sum_{r\in\mathcal{R}} |r\rangle_R \otimes \sqrt{M^{AB}_{r,\gamma}} \otimes |\gamma\rangle_C$.

A.5 Feasible Set and Channel Model

In the asymptotic limit, we define the feasible set as the set of density matrices satisfying our observations. Thus, it becomes apparent that we would need some means to simulate these observations.

First, let us define the feasible sets for prepare-and-measure and entanglement based protocols. In the asymptotic limit one finds

$$S_{EB} := \{\rho_{AB} \in \mathrm{Pos}(\mathcal{H}_A \otimes \mathcal{H}_B) | \operatorname{Tr}\left[\left(M_k^A \otimes M_l^B\right)\rho_{AB}\right] = \gamma_{kl}, \operatorname{Tr}[\rho_{AB}] = 1\}, \tag{A.38}$$

$$\begin{aligned} S_{PM} := \{\rho_{AB} \in \mathrm{Pos}(\mathcal{H}_A \otimes \mathcal{H}_B) | \operatorname{Tr}\left[\left(M_k^A \otimes M_l^B\right)\rho_{AB}\right] = \gamma_{kl}, \\ \operatorname{Tr}[\rho_{AB}] = 1, \operatorname{Tr}_B[\rho_{AB}] = \operatorname{Tr}_{A'}\left[|\psi\rangle\langle\psi|_{AA'}\right]\}. \end{aligned} \tag{A.39}$$

The above sets possess this form because in the asymptotic regime, we use a point-set in the acceptance test. In the finite-size regime, one would use a set which does not accept on a single point only. More information on acceptance tests can be found in [19] under its old name parameter estimation. However, we will discuss this in a separate guide.

The last constraint in the prepare-and-measure set is due to the source-replacement scheme as described in Appendix A.3.2. In principle, the values for each γ_{kl} could be taken from experimental data, but then this would no longer be an asymptotic protocol, and finite-size effects must be considered. This is possible in our software, but will be covered in a different guide.

For our asymptotic key rate calculation, we simulate γ_{kl} with a channel model. Assuming a particular channel model $\Phi : AA' \to AB$, the appropriate observations γ_{kl} are then found by applying Alice's and Bob's POVM elements onto ρ_{AB},

$$\operatorname{Tr}\left[\left(M_k^A \otimes M_l^B\right)\Phi(\rho_{AA'})\right] = \gamma_{kl}. \tag{A.40}$$

The software provides a few prepackaged channel models and tools, but the user can supplement them with their own. The example A.5.1 shows how we simulate the statistics of a depolarizing and misalignment channel.

Example A.5.1: BB84 Depolarizing and Misalignment Channel

Here we construct a depolarizing and misalignment channel. The depolarization is parameterized by λ ranging from 0 (none) to 1 (fully) and misalignment is considered around the y-axis in radians.
The action of a channel can be simulated by starting with $\rho_{AA'}$ and calculating the effect of Φ on it. First, the depolarizing channel Δ_λ acts on a density matrix $\sigma_{A'}$ as

$$\Delta_\lambda(\sigma_{A'}) = (1-\lambda)\sigma_{A'} + \lambda\frac{I_{A'}}{\dim(A')} \tag{A.41}$$

where $\lambda \in [0,1]$ is the depolarization parameter and $I_{A'}$ is the maximally mixed state. Since we consider the prepare-and-measure version of the BB84 protocol we need to apply Δ_λ to A' only, and in order to do this, we use the Kraus representation of the channel. The Kraus operators are

$$K_0 = \sqrt{1-\frac{3\lambda}{4}}I, \quad K_1 = \frac{\sqrt{\lambda}}{2}Z, \quad K_2 = \frac{\sqrt{\lambda}}{2}X, \quad K_3 = \frac{\sqrt{\lambda}}{2}Y. \tag{A.42}$$

Next, misalignment around the y-axis is given by the unitary rotation

$$U(\theta) = \cos(\theta)I - i\sin(\theta)Y = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{A.43}$$

In summary, the full channel is thus given by

$$\Phi(\rho_{AA'}) = \sum_i (I_A \otimes K_i U)\rho_{AA'}(I_A \otimes U^\dagger K_i^\dagger). \tag{A.44}$$

In the final step we then use our POVM elements from example A.3.1 and calculate the trace to find the observations. For only depolarization and no misalignment, one would expect the results in table A.1 below for the observations γ_{kl}. The parameter $\frac{\lambda}{2}$ plays the role of an error rate in these observations.

γ_{kl}	0	1	+	-
0	$\left(1-\frac{\lambda}{2}\right)\frac{p_z^2}{2}$	$\frac{\lambda}{2}\frac{p_z^2}{2}$	$\frac{p_z p_x}{4}$	$\frac{p_z p_x}{4}$
1	$\frac{\lambda}{2}\frac{p_z^2}{2}$	$\left(1-\frac{\lambda}{2}\right)\frac{p_z^2}{2}$	$\frac{p_z p_x}{4}$	$\frac{p_z p_x}{4}$
2	$\frac{p_z p_x}{4}$	$\frac{p_z p_x}{4}$	$\left(1-\frac{\lambda}{2}\right)\frac{p_x^2}{2}$	$\frac{\lambda}{2}\frac{p_x^2}{2}$
3	$\frac{p_z p_x}{4}$	$\frac{p_z p_x}{4}$	$\frac{\lambda}{2}\frac{p_x^2}{2}$	$\left(1-\frac{\lambda}{2}\right)\frac{p_x^2}{2}$

Table A.1: Joint probability distribution for Alice $(0, 1, 2, 3)$ and Bob $(0, 1, +, -)$ for only depolarization and no misalignment. Here $p_x = 1 - p_z$.

A.6 Error Correction

The final piece left in Eq. (A.1) is the error correction cost δ_{leak}. It is calculated by

$$\delta_{\text{leak}} = f_{\text{EC}}\, \mathrm{H}\left(R \,|\, YC\right), \tag{A.45}$$

where f_{EC} is a correction factor for non-ideal error correction. Typical values for f_{EC} are greater than one, e.g. $f_{\text{EC}} = 1.2$. In general, $f_{\text{EC}} = 1$ would correspond to perfect error correction at the Shannon limit. Furthermore, note that we only perform error correction on the signals which survive sifting and after the key map is applied by the corresponding party.

First, we recast the conditional entropy by conditioning on the public announcements

$$\delta_{\text{leak}} = f_{\text{EC}} \sum_{\nu \in \mathcal{C}_{\text{sift}}} p(\nu)\, \mathrm{H}\left(R \,|\, Y; C = \nu\right), \tag{A.46}$$

where $\mathcal{C}_{\text{sift}} \subseteq \mathcal{C}$ labels the public announcements which survive sifting. Hence, the following two probability distributions are needed in order to calculate the error correction cost

$$\text{gain}_\nu := p(\nu), \qquad \text{and} \tag{A.47}$$

$$p(r, y|\nu). \tag{A.48}$$

Based on these probability distributions, our software calculates the error correction cost automatically. Next, we show an example A.6.1, where one can calculate the error correction cost theoretically.

Example A.6.1: BB84 Error Correction Cost

To handle the error correction cost, we have to work it out for each announcement that survived sifting. For the announcements that survived sifting. Recall the error correction cost is given by

$$\delta_{\text{leak}} = f_{\text{EC}} \sum_{\nu \in \mathcal{C}_{\text{sift}}} p(\nu)\, \mathrm{H}\left(R \,|\, Y; C = \nu\right). \tag{A.49}$$

Based on the announcements in our example here, this amounts to

$$\delta_{\text{leak}} = f_{\text{EC}} \sum_{\substack{\alpha \in \mathcal{C}_A, \beta \in \mathcal{C}_B \\ \text{s.t. } \alpha = \beta}} p(\alpha, \beta)\, \mathrm{H}\left(R \,|\, Y; C_A = \alpha, C_B = \beta\right) \tag{A.50}$$

Let us continue with the example for depolarization only as in Table A.1. We first compute the gains for our announcements. Using table A.1, the gains are

$$\begin{aligned} \text{gain}_Z = p(\alpha = \beta = Z) = 2\left(1 - \frac{\lambda}{2}\right)\frac{p_z^2}{2} + 2\frac{\lambda}{2}\frac{p_z^2}{2} = p_z^2, \\ \text{gain}_X = p(\alpha = \beta = X) = 2\left(1 - \frac{\lambda}{2}\right)\frac{p_x^2}{2} + 2\frac{\lambda}{2}\frac{p_x^2}{2} = p_x^2 = (1 - p_z)^2. \end{aligned} \tag{A.51}$$

Next, we need to renormalize our data from Table A.1 conditioned on each announcement. Splitting the result in two separate tables for each basis, gives us the new tables A.2a and A.2b.

$p(r, y \vert \alpha = \beta = Z)$	$y = 0$	$y = 1$
$r = 0$	$\frac{1}{2}\left(1 - \frac{\lambda}{2}\right)$	$\frac{1}{2}\frac{\lambda}{2}$
$r = 1$	$\frac{1}{2}\frac{\lambda}{2}$	$\frac{1}{2}\left(1 - \frac{\lambda}{2}\right)$

(a) Conditioned on both Alice and Bob measuring in Z-basis.

$p(r, y \vert \alpha = \beta = X)$	$y = 0$	$y = 1$
$r = 0$	$\frac{1}{2}\left(1 - \frac{\lambda}{2}\right)$	$\frac{1}{2}\frac{\lambda}{2}$
$r = 1$	$\frac{1}{2}\frac{\lambda}{2}$	$\frac{1}{2}\left(1 - \frac{\lambda}{2}\right)$

(b) Conditioned on both Alice and Bob measuring in X-basis.

Table A.2: Joint probability distributions for the key r and Bob y conditioned on the announcements α, β.

Finally, one can calculate the conditional entropies for each announcement separately and add them up according to Eq. (A.50). The methods presented here still work in exactly the same manner if one considers the lossy situation. We find for both cases for the conditional entropy

$$\mathrm{H}\left(R \,|\, Y; \alpha = \beta = Z\right) = \mathrm{H}\left(R \,|\, Y; \alpha = \beta = X\right) = h\left(\frac{\lambda}{2}\right), \tag{A.52}$$

where $h(\cdot)$ is the binary entropy function. Hence, the final error correction leakage is given by

$$\delta_{\text{leak}} = \left(p_z^2 + (1 - p_z)^2\right) f_{\text{EC}} h\left(\frac{\lambda}{2}\right). \tag{A.53}$$

Again, the parameter $\frac{\lambda}{2}$ plays the role of an error rate here.

Again, our software simplifies this process significantly, with tools to calculate the error correction cost. The only terms which need to be specified for this are the gains (the probabilities of getting each announcement ν_i), which in this case are $\texttt{gain}_Z$, and $\texttt{gain}_X$. Furthermore, one needs to supply the conditional probability distributions $p(x, y|\nu_i)$, which in our example here are tables A.2a and A.2b.

A.7 Reducing the Dimensions of Registers

A.7.1 Reducing the Dimensions in the $\mathcal{G}$ and $\mathcal{Z}$ map via Isometries

This section details some useful tricks to reduce the dimensions of the numerical problem that needs to be optimised. Reducing the dimensions results in both faster run times and higher precision. The tricks presented here might not always be applicable and depend on the specific protocol. We first make note of the following property of the relative entropy,

$$\mathrm{D}\left(\rho \,\|\, \sigma\right) = \mathrm{D}\left(V\rho V^\dagger \,\middle\|\, V\sigma V^\dagger\right), \tag{A.54}$$

for any isometry V. We can see that after we have filtered our announcements, we might be repeating information in our announcement register C. We can write a simple isometry, $V_C = \sum_\alpha |\alpha\alpha\rangle_C \langle\alpha|_{\bar{C}}$ that transforms a single copy to two copies. We used $\bar{C}$ to label the system of reduced size. Using this we can reduce the dimensions of the Kraus operators of the $\mathcal{G}$-map in Eq. (A.21). The new Kraus operators K' are related to the old ones K by

$$K' = V_C^\dagger K. \tag{A.55}$$

Example A.7.1: BB84 Reducing Dimensions in $\mathcal{G}$ and $\mathcal{Z}$ map I

Applying the isometry V_C described above we can simplify our Kraus operators from Eq. (A.32) in example A.4.1 to

$$\begin{aligned} K'_Z &= \sqrt{p_z}\left(|0\rangle_R \otimes |0\rangle\langle 0|_A + |1\rangle_R \otimes |1\rangle\langle 1|_A\right) \otimes I_B \otimes |0\rangle_{\bar{C}}\,, \\ K'_X &= \sqrt{1-p_z}\left(|0\rangle_R \otimes |2\rangle\langle 2|_A + |1\rangle_R \otimes |3\rangle\langle 3|_A\right) \otimes I_B \otimes |1\rangle_{\bar{C}}\,, \end{aligned} \tag{A.56}$$

and note again that $K'_0 = V_C^\dagger K_{0,0}$, and $K'_1 = V_C^\dagger K_{1,1}$.
The new projection operators for the $\mathcal{Z}$-map from example A.4.2 simply drop one of the dimensions from C since they now act on $\bar{C}$.

Furthermore, one can apply another isometry on Alice and the key register in the form of

$$V_{RA} = \sum_{\substack{i\in\mathcal{X}_\alpha \\ \alpha\in\mathcal{C}_A}} |ii\rangle_{RA} \langle i|_R \otimes |\alpha\rangle\langle\alpha|_{\bar{C}}\,. \tag{A.57}$$

Intuitively, this removes the register A from the Kraus operators and directly stores the result of the key map in register R. Again, the new Kraus operators K' are related to the original ones K by

$$K' = V_{RA}^\dagger K. \tag{A.58}$$

Example A.7.2: BB84 Reducing Dimensions in $\mathcal{G}$ and $\mathcal{Z}$ map II

We continue from the previous example A.7.1 and reduce the dimension of the Kraus operators even further. Applying the isometry V_{RA} to the Kraus operators K_i' of example A.7.1 simplifies them to

$$\begin{aligned} K_Z'' &= \sqrt{p_z}\left(\left|0\right\rangle_R \left\langle 0\right|_A + \left|1\right\rangle_R \left\langle 1\right|_A\right) \otimes I_B \otimes \left|0\right\rangle_{\bar{C}} , \\ K_X'' &= \sqrt{1-p_z}\left(\left|0\right\rangle_R \left\langle 2\right|_A + \left|1\right\rangle_R \left\langle 3\right|_A\right) \otimes I_B \otimes \left|1\right\rangle_{\bar{C}} . \end{aligned} \tag{A.59}$$

A.7.2 Reducing Source-Replacement Dimension via Schmidt Decomposition

Another technique to reduce the dimensions of the optimisation problem is the Schmidt decomposition of the initial state $\left|\psi\right\rangle_{AA'}$ of Eq. (A.6). Starting from our previous description, Eq. (A.6), we can write Alice's sent state as

$$\left|\psi\right\rangle_{AA'} = \sum_{i=1}^{d_A} \sqrt{p_i}\left|i\right\rangle_A \left|s_i\right\rangle_{A'} . \tag{A.60}$$

Applying the Schmidt decomposition and identifying $m := \min\{\dim(A), \dim(A')\}$ lets us write it as

$$\left|\psi\right\rangle_{AA'} = \sum_{k=1}^{m} \alpha_k \left|u_k\right\rangle_A \left|v_k\right\rangle_{A'} , \tag{A.61}$$

where $\alpha_k \geq 0$ for all k. The $\{\left|u_k\right\rangle\}$, $\{\left|v_k\right\rangle\}$ and $\{\alpha_k\}$ can be found by a singular value decomposition. Define the projector $\Pi_A = \sum_{k=1}^{m} \left|u_k\middle\rangle\middle\langle u_k\right|$ acting on Alice's POVM elements. Then, Alice's POVM elements simplify to

$$M_k^A = \Pi_A \left|k\middle\rangle\middle\langle k\right| \Pi_A \quad \forall k. \tag{A.62}$$

Example A.7.3: BB84 Reducing Dimensions in $\mathcal{G}$ and $\mathcal{Z}$ map III

The Schmidt decomposition lets us use a significantly smaller dimension for Alice, and we can rewrite $\left|\psi\right\rangle_{AA'}$ of example A.3.1 as

$$\left|\psi\right\rangle_{AA'} = \frac{1}{\sqrt{2}}\left(\left|00\right\rangle_{AA'} + \left|11\right\rangle_{AA'}\right). \tag{A.63}$$

Then, Alice's POVM becomes

$$M_A' = \{p_z \left|0\middle\rangle\middle\langle 0\right|, p_z \left|1\middle\rangle\middle\langle 1\right|, (1-p_z)\left|+\middle\rangle\middle\langle +\right|, (1-p_z)\left|-\middle\rangle\middle\langle -\right|\}. \tag{A.64}$$

Using these POVM elements to construct the Kraus operators of the $\mathcal{G}$ map as in example A.4.1 together with our previous reductions, example A.7.1 and example A.7.2 results in

$$\begin{aligned} K_Z''' &= p_z\left(\left|0\right\rangle_R \left\langle 0\right|_A + \left|1\right\rangle_R \left\langle 1\right|_A\right) \otimes I_B \otimes \left|0\right\rangle_{\bar{C}} , \\ K_X''' &= (1-p_z)\left(\left|0\right\rangle_R \left\langle +\right|_A + \left|1\right\rangle_R \left\langle -\right|_A\right) \otimes I_B \otimes \left|1\right\rangle_{\bar{C}} . \end{aligned} \tag{A.65}$$

Appendix B

Primal and Dual SDP for Linearization

In this section we cover the general form of the primal and dual problem which we solve after linearization from the Frank-Wolfe 2 step-solver. For this section, all vector/inner product/Hilbert spaces are assumed to be finite dimensional, over the field $\mathbb{R}$, and are self-dual. All sets are also assumed to be closed. Furthermore, we use $\mathbb{R}_+$ to denote the non-negative real numbers.

B.1 Background Definitions

This section is only meant as a reference for readers. It only covers the definitions required to verify the primal and dual problems we use. The majority of the definitions were taken from [6, 20].

Here we give the basic definitions of convex sets and functions, which form the foundational objects we use.

Definition 1. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and S a subset of $\mathcal{H}$. S is called a* convex set *if for all $x, y \in S$ and $\theta \in [0,1]$,*

$$\theta x + (1-\theta)y \in S. \tag{B.1}$$

Definition 2. *Let $\mathcal{H}$ be a Hilbert space and $S \subset \mathcal{H}$ a be a convex set. Then, a function $f : S \to \mathbb{R}$ is called a* convex function *(on S) if for all $x, y \in S$ and $\theta \in [0,1]$,*

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y). \tag{B.2}$$

Similarly, f is called a concave function *(on S) if $-f$ is convex.*

Note that for a convex (concave) function f, the set of points above f defined as $\{x \oplus y \in S \oplus \mathbb{R} \,|\, y \geq f(x)\}$ (the set of points below f defined as $\{x \oplus y \in S \oplus \mathbb{R} \,|\, y \leq f(x)\}$), is a convex set. This will help us build convex sets out of convex and concave functions. Of particular importance to us are a type of convex set called convex cones, which will form the core components of our optimization problem's constraints.

Definition 3. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and $\mathcal{K}$ a subset of $\mathcal{H}$. $\mathcal{K}$ is called a* convex cone *if it satisfies the following,*

1. *$\mathcal{K}$ is a convex set.*
2. *For all $x \in \mathcal{K}$ and $\lambda \in \mathbb{R}_+$, $\lambda x \in \mathcal{K}$.*

Some important examples of convex cones are positive semi-definite operators $\mathrm{Pos}(\mathcal{H}) \subset \mathrm{Herm}(\mathcal{H})$, and the n-dimensional positive orthant $\mathbb{R}^n_+$. Furthermore, the triangle inequality makes all norms convex functions, and we use these norms to construct convex cones as follows.

Definition 4. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and let $\|\cdot\|_a : \mathcal{H} \to \mathbb{R}$ be a norm on $\mathcal{H}$. We define and denote the* norm cone *on the Hilbert space $\mathcal{H} \oplus \mathbb{R}$ as,*

$$\mathcal{K}_{\|\cdot\|_a} := \left\{ x \oplus t \in \mathcal{H} \oplus \mathbb{R} \,|\, \|x\|_a \leq t \right\}. \tag{B.3}$$

Before we give the optimization problem, we note that there exists a closely associated optimization problem called the dual problem which will help us solve the original optimization. Before we can give the definitions of the primal and dual problem we first have to define what we mean by "dual" in this context. Specifically, we focus on dual norms, and dual cones.

Definition 5. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and $\mathcal{K}$ a cone in $\mathcal{H}$. We define the* dual cone[1] *of $\mathcal{K}$ as*

$$\mathcal{K}' := \left\{ y \in \mathcal{H} \,|\, \forall x \in \mathcal{K}, \langle y, x \rangle \geq 0 \right\}. \tag{B.4}$$

For the dual of norm cones, we first define the dual norm.

Definition 6. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and be a $\|\cdot\|_a : \mathcal{H} \to \mathbb{R}$ norm on $\mathcal{H}$. We denote and define the* dual norm, *$\|\cdot\|_{a'}$, of $\|\cdot\|_a$ as*

$$\|y\|_{a'} = \sup \left\{ \langle y, x \rangle \,|\, x \in \mathcal{H}, \|x\|_a \leq 1 \right\}. \tag{B.5}$$

Notably for us, the vector norms $\|\cdot\|_1$, and $\|\cdot\|_\infty$ are dual to each other, and similarly the Schatten spectral type norms for matrices $\|\cdot\|_1$ and $\|\cdot\|_\infty$ are also dual to each other. From this, we can construct the dual of norm cones.

Theorem 1. *Let $\mathcal{H}$ be a Hilbert space over the field $\mathbb{R}$, and $\|\cdot\|_a : \mathcal{H} \to \mathbb{R}$ be a norm on $\mathcal{H}$. Using the definition of the dual cone, the dual cone of $\mathcal{K}_{\|\cdot\|_a}$ is given by,*

$$\mathcal{K}'_{\|\cdot\|_a} = \mathcal{K}_{\|\cdot\|_{a'}}. \tag{B.6}$$

The curious reader may have noted that we kept referring to our dual cones as "dual to each other", in fact we can extend this for all convex cones.

Theorem 2. *For all convex cones $\mathcal{K} \subset \mathcal{H}$, it holds that*

$$\mathcal{K}'' := (\mathcal{K}')' = \mathcal{K}. \tag{B.7}$$

A rather interesting example is if $\mathcal{K} = \mathcal{H}$, then $\mathcal{K}' = \{\vec{0}\}$, where $\vec{0}$ denotes the zero vector. Other special cases include self dual convex cones.

[1] This is not to be confused with the dual of a Hilbert space.

Definition 7. *The convex cone $\mathcal{K}$ is called* self dual *if $\mathcal{K}' = \mathcal{K}$.*

For us, some important self dual convex cones are $\mathbb{R}^n_+$ and $\mathrm{Pos}(\mathcal{H}) \subset \mathrm{Herm}(\mathcal{H})$ for any finite dimensional Hilbert space.

Finally, we have enough to define conic optimization problems.

Definition 8. *Let A and B be Hilbert spaces over the field $\mathbb{R}$. Let $\mathcal{K}_A$ and $\mathcal{K}_B$ be convex cones in A and B respectively. Let $F_A \in A$ and $F_B \in B$. Let $\mathcal{N} : A \to B$ be a linear map with adjoint map $\mathcal{N}^\dagger$. We define the* primal conic problem *and its associated* dual conic problem *by the following*[2]

Primal:

$$\begin{aligned} \underset{\rho_A}{\text{minimize:}} \quad & \langle \rho_A, F_A \rangle \\ \text{subject to:} \quad & \mathcal{N}(\rho_A) - F_B \in \mathcal{K}_B \\ & \rho_A \in \mathcal{K}_A. \end{aligned} \tag{B.8}$$

Dual:

$$\begin{aligned} \underset{\sigma_B}{\text{maximize:}} \quad & \langle \sigma_B, F_B \rangle \\ \text{subject to:} \quad & F_A - \mathcal{N}^\dagger(\sigma_B) \in \mathcal{K}'_A \\ & \sigma_B \in \mathcal{K}'_B. \end{aligned} \tag{B.9}$$

B.2 General Conic Optimization

Instead of taking the dual of only our exact problem, we will determine the dual to a full class of problems, and show that our problem is a member of that class. Let $n, m \in \mathbb{Z}_+$ and $A, B_1 \ldots B_n, C_1 \ldots C_m$ be Hilbert spaces over the field $\mathbb{R}$. Furthermore, let $(\|\cdot\|_{a_i}, \Phi_i, \sigma_i, \mu_i)|_{i=1}^n$ be quartets denoting constraints, where

1. $\|\cdot\|_{a_i} : B_i \to \mathbb{R}$ is a norm,
2. $\Phi_i : A \to B_i$ is a linear map with dual $\Phi_i^\dagger$,
3. $\sigma_i \in B_i$,
4. $\mu_i \in \mathbb{R}_+$.

Additionally, let $(\Psi_j, \tau_j, \mathcal{K}_{C_j})|_{j=1}^m$ be triplets denoting constraints where

1. $\Psi_j : A \to C_j$ is a linear map with dual $\Psi_j^\dagger$.
2. $\tau_j \in C_j$.
3. $\mathcal{K}_{C_j}$ is a cone in C_j.

[2] The minimization and maximization use inf and sup respectively, however numerically solvable problems are typically restricted to sets where we can swap these for min and max.

Finally, let $F_A \in A$, and $\mathcal{K}_A \subset A$ be a self-dual cone, i.e. $\mathcal{K}'_A = \mathcal{K}_A$. Then, we define our primal as

$$\begin{aligned} \underset{\rho}{\text{minimize:}} \quad & \langle \rho, F_A \rangle \\ \text{subject to:} \quad & \|\Phi_i(\rho) - \sigma_i\|_{a_i} \leq \mu_i, \; i = 1, \ldots, n, \\ & \Psi_j(\rho) - \tau_j \in \mathcal{K}_{C_j}, \; j = 1, \ldots, m, \\ & \rho \in \mathcal{K}_A. \end{aligned} \tag{B.10}$$

To rewrite our SDP in our general conic optimization form, we first recast our optimization variable and constraints as elements of norm cones. To achieve this, we look at how to rewrite norm balls through affine maps on norm cones.

Theorem 3. *Let $\mathcal{B}$ be the norm ball, $\mathcal{B} := \{\rho \in \mathcal{H} \,|\, \|\rho\|_a \leq \mu\}$, and let $\Theta : \mathcal{H} \to \mathcal{H} \oplus \mathbb{R}$ be the affine transformation given by, $\Theta(\rho) = (\rho \oplus 0) - (0 \oplus -\mu)$, then $\rho \in \mathcal{B}$ if and only if $\Theta(\rho) \in \mathcal{K}_{\|\cdot\|_a}$.*

We can further extend this to any constraint of the form,

$$\|\mathcal{E}(\rho) - \sigma\|_a \leq \mu, \tag{B.11}$$

where $\mathcal{E}$ is any linear map, and get the new constraint,

$$(\mathcal{E}(\rho) \oplus 0) - (\sigma \oplus -\mu) \in \mathcal{K}_{\|\cdot\|_a}. \tag{B.12}$$

Applying this to all the norm based constraints in Eq. (B.10) gives the equivalent primal problem,

$$\begin{aligned} \underset{\rho}{\text{minimize:}} \quad & \langle \rho, F_A \rangle \\ \text{subject to:} \quad & (\Phi_i(\rho) \oplus 0) - (\sigma_i \oplus -\mu_i) \in \mathcal{K}_{\|\cdot\|_{a_i}}, \; i = 1, \ldots, n, \\ & \Psi_j(\rho) - \tau_j \in \mathcal{K}_{C_j}, \; j = 1, \ldots, m, \\ & \rho \in \mathcal{K}_A. \end{aligned} \tag{B.13}$$

Our general form for primal and dual conic problems in Eqs. (B.8) and (B.9) only used two Hilbert spaces ($\mathcal{H}_1$, $\mathcal{H}_2$), two convex cones ($\mathcal{K}_1$, $\mathcal{K}_2$), two constants (F_1, F_2), and a single linear map ($\mathcal{N} : \mathcal{H}_1 \to \mathcal{H}_2$). We concatenate Hilbert spaces, cones, variables, constraints and affine maps to fit this format. Below we present a summary of these, along with their dual cones and dual map.

1. The Hilbert spaces $\mathcal{H}_1$ is still just A for ρ. Hilbert space $\mathcal{H}_2$ is the concatenation of all the output spaces from the constraints in Eq. (B.13).

$$\mathcal{H}_1 := A, \quad \mathcal{H}_2 := \bigoplus_{i=1}^{n} (B_i \oplus \mathbb{R}) \oplus \bigoplus_{j=1}^{m} C_j. \tag{B.14}$$

2. The convex cone $\mathcal{K}_1$ is the convex cone where ρ is drawn from, while $\mathcal{K}_2$ is formed from all $\mathcal{K}_{\|\cdot\|_{a_i}}$ and $\mathcal{K}_{C_j}$.

$$\mathcal{K}_1 := \mathcal{K}_A, \quad \mathcal{K}_2 := \bigoplus_{i=1}^{n} \mathcal{K}_{\|\cdot\|_{a_i}} \oplus \bigoplus_{j=1}^{m} \mathcal{K}_{C_j}. \tag{B.15}$$

The dual cones of $\mathcal{K}_1$ and $\mathcal{K}_2$ are therefore the duals of the cones that compose them. Also recall that we defined $\mathcal{K}_A$ to be self dual.

$$\mathcal{K}'_1 = \mathcal{K}_A = \mathcal{K}_1, \quad \mathcal{K}'_2 = \bigoplus_{i=1}^{n} \mathcal{K}_{\|\cdot\|_{a'_i}} \oplus \bigoplus_{j=1}^{m} \mathcal{K}'_{C_j}, \tag{B.16}$$

3. We gather together the constant terms in the objective for F_1 and all the constants in the affine expressions in the constraints for F_2.

$$F_1 := F_A, \quad F_2 := \bigoplus_{i=1}^{n}(\sigma_i \oplus -\mu_i) \oplus \bigoplus_{j=1}^{m} \tau_j. \tag{B.17}$$

4. The linear transformation $\mathcal{N}$ is constructed by combining the linear maps $\Phi_i(\rho)$, and $\Psi_j(\rho)$ together into a single linear map.

$$\mathcal{N}(\rho) := \bigoplus_{i=1}^{n}(\Phi_i(\rho) \oplus 0) \oplus \bigoplus_{j=1}^{m} \Psi_j(\rho). \tag{B.18}$$

The dual map is constructed in the usual way from the definition $\left\langle \mathcal{N}^\dagger(\sigma), \rho \right\rangle = \langle \sigma, \mathcal{N}(\rho) \rangle$, $\forall \rho, \sigma$. Gathering like terms gives,

$$\mathcal{N}^\dagger \left(\bigoplus_{i=1}^{n}(\alpha_i \oplus s_i) \oplus \bigoplus_{j=1}^{m} \beta_j \right) = \sum_{i=1}^{n} \Phi_i^\dagger(\alpha_i) + \sum_{j=1}^{m} \Psi_j^\dagger(\beta_j) + 0\vec{s}. \tag{B.19}$$

Using these dual cones and dual maps, we construct the dual problem,

$$\begin{aligned} \underset{\substack{\alpha_1,\ldots,\alpha_n \\ s_1,\ldots,s_n \\ \beta_1,\ldots,\beta_m}}{\text{maximize:}} \quad & \left\langle \bigoplus_{i=1}^{n}(\alpha_i \oplus s_i) \oplus \bigoplus_{j=1}^{m} \beta_j, \bigoplus_{i=1}^{n}(\sigma_i \oplus -\mu_i) \oplus \bigoplus_{j=1}^{m} \tau_j \right\rangle \\ \text{subject to:} \quad & \left(F_A - \sum_{i=1}^{n} \Phi_i^\dagger(\alpha_i) - \sum_{j=1}^{m} \Psi_j^\dagger(\beta_j) - 0\vec{s} \right) \in \mathcal{K}_A, \\ & \alpha_i \oplus s_i \in \mathcal{K}_{\|\cdot\|_{a_i'}}, \; i = 1, \ldots, n, \\ & \beta_j \in \mathcal{K}'_{C_j}, \; j = 1, \ldots, m. \end{aligned} \tag{B.20}$$

This can be simplified into the final dual problem,

$$\begin{aligned} \underset{\substack{\alpha_1,\ldots,\alpha_n, \vec{s} \\ \beta_1,\ldots,\beta_m}}{\text{maximize:}} \quad & \sum_{i=1}^{n} \langle \alpha_i, \sigma_i \rangle + \sum_{j=1}^{m} \langle \beta_j, \tau_j \rangle - \langle \vec{\mu}, \vec{s} \rangle \\ \text{subject to:} \quad & F_A - \sum_{i=1}^{n} \Phi_i^\dagger(\alpha_i) - \sum_{j=1}^{m} \Psi_j^\dagger(\beta_j) \in \mathcal{K}_A, \\ & \alpha_i \in B_i, \|\alpha_i\|_{a_i'} \leq s_i, \; i = 1, \ldots, n, \\ & \beta_j \in \mathcal{K}'_{C_j}, \; j = 1, \ldots, m, \\ & \vec{s} \in \mathbb{R}^n_+, \end{aligned} \tag{B.21}$$

where $\vec{\mu} = [\mu_1 \ldots \mu_n]^{\mathrm{T}}$.

B.3 Application to the Linearization

Due to Eq. (A.1) the goal is to minimize the relative entropy for all ρ_{AB} that satisfy our constraints from Section 3.2.8. Unfortunately, the quantum relative entropy function is currently not supported by the majority of disciplined convex solvers. Instead we use methods such as the Frank-Wolfe algorithm [21] to solve a series of convex linearizations.

Let us define

$$f(\rho_{AB}) := \mathrm{D}\left(\mathcal{G}\left(\rho_{AB}\right) \| \, \mathcal{Z} \circ \mathcal{G}\left(\rho_{AB}\right)\right)^{3}, \tag{B.22}$$

with gradient

$$\boldsymbol{\nabla} f(\rho_{AB}) = \mathcal{G}^{\dagger}\left(\log \circ \mathcal{G}(\rho_{AB}) - \log \circ \mathcal{Z} \circ \mathcal{G}(\rho_{AB})\right), \tag{B.23}$$

and let $\boldsymbol{\nabla} f(\rho_{AB}^{(0)})$ be the gradient evaluated at the point $\rho_{AB}^{(0)}$. We then have to solve SDPs of the form

$$\begin{aligned} \underset{\rho_{AB}}{\text{minimize:}} \quad & \left\langle \rho_{AB}, \boldsymbol{\nabla} f(\rho_{AB}^{(0)}) \right\rangle \\ \text{subject to:} \quad & \rho_{AB} \in S, \\ & \rho_{AB} \geq 0, \end{aligned} \tag{B.24}$$

where S represents a list of convex constraints on ρ_{AB}.

Each constraint from S is written as either an affine equality, affine inequality, vector 1-norm, or matrix 1-norm constraint. For each of these constraint types, we show how to rewrite them into the form used by the primal in Eq. (B.10), and convert them to their form in the dual problem in Eq. (B.21). Additionally, we show some small simplifications to the dual.

1. **Equality:** Let $\Gamma_{AB} \in \mathrm{Herm}(AB)$ and $\gamma \in \mathbb{R}$. We get the constraint,

$$\mathrm{Tr}[\Gamma_{AB}\rho_{AB}] = \gamma \pm \mathrm{tol.}, \tag{B.25}$$

and convert to the general form,

$$\begin{gathered} \Phi_{AB\to\mathbb{R}}(\rho_{AB}) = \mathrm{Tr}[\Gamma_{AB}\rho_{AB}], \\ \sigma = \gamma, \\ \mu = \mathrm{tol.}, \\ \left\|\Phi_{AB\to\mathbb{R}}(\rho_{AB}) - \sigma\right\|_1 \leq \mu. \end{gathered} \tag{B.26}$$

The map Φ, has the simple dual $\Phi^{\dagger}(\alpha) = \Gamma_{AB}\alpha$. In the dual problem, we gain the constraint,

$$\begin{gathered} \boldsymbol{\nabla} f(\rho_{AB}^{(0)}) + \cdots + (-\Gamma_{AB}\alpha) + \cdots \geq 0, \\ \|\alpha\|_{\infty} \leq s, \\ s \in \mathbb{R}_{+},\ \alpha \in \mathbb{R}, \end{gathered} \tag{B.27}$$

and the objective picks up the terms,

$$\cdots + \gamma\alpha + \cdots - \mathrm{tol.}s - \ldots. \tag{B.28}$$

Further more, $\|\alpha\|_{\infty} = |\alpha|$, and we can convert $\|\alpha\|_{\infty} \leq s$ to $-s \leq \alpha \leq s$. Given multiple such dual constraints we collect them into the vector format $-\vec{s} \leq \vec{\alpha} \leq \vec{s}$.

[3] Typically, we use the perturbed function $f_{\epsilon}(\rho_{AB})$ from [1] with our corrections from Appendix B.4.2 in its place.

2. **Inequality:** Let $\Gamma_{AB} \in \mathrm{Herm}(AB)$ and let $\gamma_L, \gamma_U \in \mathbb{R}$ with $\gamma_L \leq \gamma_U$. We get the constraint,

$$\gamma_L - \text{tol.} \leq \mathrm{Tr}[\Gamma_{AB}\rho_{AB}] \leq \gamma_U + \text{tol.}, \tag{B.29}$$

and convert to the general form,

$$\begin{gathered} \Phi_{AB\to\mathbb{R}}(\rho_{AB}) = \mathrm{Tr}[\Gamma_{AB}\rho_{AB}], \\ \sigma = \frac{\gamma_U + \gamma_L}{2}, \\ \mu = \frac{\gamma_U - \gamma_L}{2} + \text{tol.}, \\ \left\|\Phi_{AB\to\mathbb{R}}(\rho_{AB}) - \sigma\right\|_1 \leq \mu. \end{gathered} \tag{B.30}$$

Following the same procedure for the equality constraints, our dual gains the constraint,

$$\begin{gathered} \boldsymbol{\nabla} f(\rho_{AB}^{(0)}) + \cdots + (-\Gamma_{AB}\alpha) + \cdots \geq 0, \\ \|\alpha\|_\infty \leq s, \\ s \in \mathbb{R}_+,\ \alpha \in \mathbb{R}, \end{gathered} \tag{B.31}$$

which are identical to the equality constraints. The objective picks up similar terms,

$$\cdots + \frac{\gamma_U + \gamma_L}{2}\alpha + \cdots - \left(\frac{\gamma_U - \gamma_L}{2} + \text{tol.}\right) s - \ldots. \tag{B.32}$$

We then apply the same process to change $\|\alpha\|_\infty \leq s$ to $-s \leq \alpha \leq s$ and collect multiple such dual constraints together into the vector format $-\vec{s} \leq \vec{\alpha} \leq \vec{s}$.

3. **Vector 1-norm:** Let $\{\Gamma_{i,AB}\}|_{i=1}^n \subset \mathrm{Herm}(AB)$, $\vec{v} \in \mathbb{R}^n$, and $\mu \in \mathbb{R}_+$.

$$\left\|\sum_{i=1}^n \mathrm{Tr}[\Gamma_i \rho_{AB}]\vec{e}_i - \vec{v}\right\|_1 \leq \mu + \text{tol.}, \tag{B.33}$$

where $\|\cdot\|_1$ is the l_1 norm on vectors. We convert this to the general constraint form,

$$\begin{gathered} \Phi_{AB\to\mathbb{R}^n}(\rho_{AB}) = \sum_{i=1}^n \mathrm{Tr}[\Gamma_{i,AB}\rho_{AB}]\vec{e}_i, \\ \sigma = \vec{v}, \\ \left\|\Phi_{AB\to\mathbb{R}^n}(\rho_{AB}) - \sigma\right\|_1 \leq \mu + \text{tol.}. \end{gathered} \tag{B.34}$$

The dual for Φ is simply $\Phi^\dagger(\vec{\alpha}) = \sum_{i=1}^n \Gamma_{i,AB}\alpha_i$. The dual problem picks up the constraint,

$$\begin{gathered} \boldsymbol{\nabla} f(\rho_{AB}^{(0)}) + \cdots + (-\sum_{i=1}^n \Gamma_{i,AB}\alpha_i) + \cdots \geq 0, \\ \|\vec{\alpha}\|_\infty \leq s, \\ s \in \mathbb{R}_+,\ \vec{\alpha} \in \mathbb{R}^n, \end{gathered} \tag{B.35}$$

and the objective function picks up the terms,

$$\cdots + \vec{v}^{\mathrm{T}}\vec{\alpha} + \cdots - (\mu + \text{tol.})\, s - \ldots. \tag{B.36}$$

we convert the condition $\|\vec{\alpha}\|_\infty \leq s$, to the equivalent set of conditions $-s \leq \alpha_i \leq s$, $\forall i$.

4. **Matrix 1-norm:** Let $\Phi_{AB\to C}$ be a hermitian preserving linear map from $\mathrm{Herm}(AB)$ to $\mathrm{Herm}(C)$, $\sigma_C \in \mathrm{Herm}(C)$, and $\mu \in \mathbb{R}_+$. We get the constraint,

$$\left\|\Phi_{AB\to C}(\rho_{AB}) - \sigma_C\right\|_1 \leq \mu + \mathrm{tol.}, \tag{B.37}$$

where $\|\cdot\|_1$ is the trace norm. This is already in the desired form so we can skip to the dual. We get the dual constraint,

$$\begin{gathered}\boldsymbol{\nabla} f(\rho_{AB}^{(0)}) + \cdots + (-\Phi^{\dagger}_{C\to AB}(\alpha_C)) + \cdots \geq 0,\\ \|\alpha_C\|_\infty \leq s,\\ s \in \mathbb{R}_+,\ \alpha_C \in \mathrm{Herm}(C),\end{gathered} \tag{B.38}$$

where $\|\cdot\|_\infty$ is the sum of singular values. The term in the objective function is simply,

$$\cdots + \mathrm{Tr}[\sigma_C\alpha_C] + \cdots - (\mu + \mathrm{tol.})s - \ldots. \tag{B.39}$$

Finally, we can rewrite $\|\alpha_C\|_\infty \leq s$ as the constraint $-sI_C \leq \alpha_C \leq sI_C$.

B.4 Proof Extensions

The math solver module `FW2StepSolver` splits the task of determining a verified lower bound on the quantum relative entropy into 2 steps. Step 1 uses the Frank-Wolfe algorithm to determine a point that upper bounds the quantum relative entropy, while step 2 uses a linearization to determine a verified lower bound [1].

In this section, we focus on extending and improving the results of theorems 2, 3 and appendix D from Ref. [1]. Improvements to step 1 of the Frank-Wolfe algorithm can improve key rate, but they do not impact the security proof [1]. Hence, improvements to step 1 will be left out of this section.

In Appendix B.4.1, we focus on improving the extension from the constraint set S to the constraint set $\tilde{S}_{\epsilon'}$ which accounted for solution precision and numerical representation in theorem 3 and appendix D from [1].

In Appendix B.4.2, we revisit the proof of [1, Theorem 2], using a more careful analysis of CPTP maps, CPTNI maps, and sub-normalized density matrices ($\mathrm{Tr}[\rho] \leq 1$). From this, we derive a better perturbation correction factor $\tilde{\zeta}_\epsilon$ and discuss better methods for choosing the perturbation factor ϵ.

Caution B.4.1: Limitations of the Numerical Proof

As in the original paper [1] there are two major areas our numerical analysis does not cover. First, errors caused by function evaluations (not covered by ϵ_{rep} in Appendix B.4.1) are implementation dependent and are hard to provide general statements for.
Second, step 2 of the Frank-Wolfe solver swaps the primal to the dual to avoid scenarios where the underlying solver cannot fully minimize the function. With the dual the solver will instead not fully maximize the function, providing a lower bound. However, this also introduces another potential source of error. All solvers are very likely to tolerate some small amount of error, especially for problems involving eigenvalues. This causes a slight expansion to the dual feasible set, which could increase the value returned from maximization. When pushing the solver to extremely low relative entropy regimes near the solver's tolerance, users should review the constraint violation information provided in `FW2StepSolver`'s debug information.

B.4.1 Finite Precision and Optimization Constraints

Reducing Imprecise Step 1 Solver Effects

Because the step 1 solver is unlikely to give a point $\rho_{AB}^{(0)}$ that is within our constraint set, appendix D.2 of [1] first shifts $\rho_{AB}^{(0)}$ to ensure the smallest eigenvalue $\lambda_{\min}$ is non-negative and within the positive semi-definite cone. They achieve this using the non-linear map,

$$\rho_{AB}^{\prime(0)} = \begin{cases} \rho_{AB}^{(0)} - \lambda_{\min} I_{AB}, & \lambda_{\min} < 0 \\ \rho_{AB}^{(0)}, & \text{otherwise} \end{cases}. \tag{B.40}$$

Following this, they expand the constraint set S to a set $S_{\epsilon_{\text{sol}}}$ so that $S \subset S_{\epsilon_{\text{sol}}}$ and $\rho_{AB}^{\prime(0)} \in S_{\epsilon_{\text{sol}}}$ and the minimization is performed over $S_{\epsilon_{\text{sol}}}$[4]. However, this last step is unnecessary, as pointed out in Section B.2 of [22]. Because $S_{\epsilon_{\text{sol}}}$ is a convex set and the relative entropy is still convex over all of $S_{\epsilon_{\text{sol}}}$, any tangent hyperplane to the surface is a lower bound on $f(\rho)$ [6]. Therefore, so long as the set we choose to minimize the linearization over contains the true optimal point for $\min_{\rho\in S} f(\rho)$, then our linearization still bounds it from below. The easiest choice is to simply minimize the linearization over the original set S instead of $S_{\epsilon_{\text{sol}}}$. This often significantly tightens the lower bound on the relative entropy.

We also apply one more small optimization. In order to allow future implementation of powerful methods such as dimension reduction [22], we have to relax the constraint $\operatorname{Tr}[\rho_{AB}] = 1$ and replace it with $\operatorname{Tr}[\rho_{AB}] \leq 1$. Applying Eq. (B.40) can now cause large shifts to the trace, providing a poor linearization point. Instead, we apply a depolarizing channel,

$$D_\epsilon(\rho_{AB}) := (1-\epsilon)\rho_{AB} + \epsilon \operatorname{Tr}[\rho_{AB}] \frac{I_{AB}}{\dim(AB)}, \tag{B.41}$$

and choose,

$$\epsilon = \frac{-\lambda_{\min}}{\operatorname{Tr}\left[\rho_{AB}^{(0)}\right] - \lambda_{\min}\dim(AB)}, \tag{B.42}$$

whenever $\lambda_{\min} < 0$. This applies the bare minimum perturbation required to raise $\lambda_{\min}$ to 0, ensuring $\rho_{AB}^{\prime(0)} \in \operatorname{Pos}(AB)$.

Tighter Bounds on Numerical Representation Error

In theorem 3 and appendix D.1 of [1], Winick et al. develop a method to handle numerical errors caused by imprecise representations stored in memory. To illustrate this problem, let $\Gamma_{AB} \in \operatorname{Herm}(AB)$ be part of the constraint $\langle \Gamma_{AB}, \rho_{AB} \rangle = \gamma$. But, the computer only stores an approximation of Γ_{AB}, we call the *finite representation* of Γ_{AB}, and denote it as $\tilde{\Gamma}_{AB}$. We also denote the difference between between these as $\delta\Gamma_{AB} := \tilde{\Gamma}_{AB} - \Gamma_{AB}$. The same goes for the scalar γ, and any other component of a constraint we must store in memory. The net effect is we optimize not over the original constraint set S but an approximate set $\tilde{S}$. Winick et al. circumvented this issue by constructing a new set $\tilde{S}_{\epsilon_{\text{rep}}}$ such that $S, \tilde{S} \subset \tilde{S}_{\epsilon_{\text{rep}}}$. Their original proof showed that for all $\rho_{AB} \in S$, all equality constraints satisfy

$$\left| \left\langle \tilde{\Gamma}_{AB}, \rho_{AB} \right\rangle - \tilde{\gamma} \right| \leq \|\delta\Gamma_{AB}\|_2 + |\delta\gamma|. \tag{B.43}$$

[4] Technically, step 2 calculates the dual problem with the dual constraints of $S_{\epsilon_{\text{sol}}}$.

Then, they picked a value ϵ_{rep} such that all the equality constraints obeyed $\|\delta\Gamma_{AB}\|_2 + |\delta\gamma| \leq \epsilon_{\text{rep}}$. They picked $\epsilon_{\text{rep}} = 10^{-10}$ which is significantly larger than MatLab's precision of $\sim 2.2 \times 10^{-16}$[5].

We improve upon their method by starting with a single estimate of numerical imprecision for scalars, ϵ_{safe}, then derive bounds for each of our constraint types. We choose $\epsilon_{\text{safe}} = 10^{-15}$, which bounds our error on individual scalar values to within an order of magnitude of Matlab's precision. The following bounds make heavy use of triangle and Holder's inequalities.

- **Equality:** We have the ideal version,

$$\begin{aligned} \langle \Gamma_{AB}, \rho_{AB} \rangle &= \gamma \\ \Leftrightarrow \langle \Gamma_{AB}, \rho_{AB} \rangle - \gamma &= 0. \end{aligned} \tag{B.44}$$

  We swap each term to its finite representation counterpart and bound them as follows,

$$\begin{aligned} \left| \left\langle \tilde{\Gamma}_{AB}, \rho_{AB} \right\rangle - \tilde{\gamma} \right| &= |\langle \delta\Gamma_{AB}, \rho_{AB} \rangle - \delta\gamma + \langle \Gamma_{AB}, \rho_{AB} \rangle - \gamma| \\ &= |\langle \delta\Gamma_{AB}, \rho_{AB} \rangle - \delta\gamma| \\ &\leq |\langle \delta\Gamma_{AB}, \rho_{AB} \rangle| + |\delta\gamma| \\ &\leq \|\delta\Gamma_{AB}\|_2 \|\rho_{AB}\|_2 + |\delta\gamma| \\ &\leq \|\delta\Gamma_{AB}\|_2 + |\delta\gamma|. \end{aligned} \tag{B.45}$$

  Because we choose ϵ_{safe} to bound the error on each scalar value, we know that $|\delta\gamma| \leq \epsilon_{\text{safe}}$. Similarly, each component of $\delta\Gamma_{AB}$ is bounded in the same manner. Using the definition of the Schatten 2-norm,

$$\|\tau\|_2 = \sqrt{\sum_{i,j} |\tau_{i,j}|^2}, \tag{B.46}$$

  we bound $\|\delta\Gamma_{AB}\|_2$ by relating it to the bounds on each component via,

$$\begin{aligned} \|\delta\Gamma_{AB}\|_2 &= \sqrt{\sum_{i,j=1}^{\dim(AB)} |\delta\Gamma_{i,j\,AB}|^2}, \\ &\leq \sqrt{\sum_{i,j=1}^{\dim(AB)} |\epsilon_{\text{safe}}|^2}, \\ &= \sqrt{\dim(AB)^2}\epsilon_{\text{safe}}, \\ &= \dim(AB)\epsilon_{\text{safe}}. \end{aligned} \tag{B.47}$$

  Our final bound is thus,

$$\left| \left\langle \tilde{\Gamma}_{AB}, \rho_{AB} \right\rangle - \tilde{\gamma} \right| \leq (\dim(AB) + 1)\, \epsilon_{\text{safe}}. \tag{B.48}$$

- **Inequality:** We have the ideal version

$$\begin{aligned} \langle \Gamma_{AB}, \rho_{AB} \rangle &\leq \gamma \\ \Leftrightarrow \langle \Gamma_{AB}, \rho_{AB} \rangle - \gamma &\leq 0. \end{aligned} \tag{B.49}$$

[5] Specifically, this is the smallest value that can be added to 1 which produces a different double precision floating point number. The exact hex value in Matlab is '3cb0000000000000'.

We swap each term to its finite representation counterpart and bound them as follows,

$$\begin{aligned}\left\langle\tilde{\Gamma}_{AB},\rho_{AB}\right\rangle-\tilde{\gamma}&=\langle\delta\Gamma_{AB},\rho_{AB}\rangle-\delta\gamma+\langle\Gamma_{AB},\rho_{AB}\rangle-\gamma,\\&\leq\langle\delta\Gamma_{AB},\rho_{AB}\rangle-\delta\gamma,\\&\leq|\langle\delta\Gamma_{AB},\rho_{AB}\rangle|+|\delta\gamma|,\\&\leq\|\delta\Gamma_{AB}\|_2\|\rho_{AB}\|_2+|\delta\gamma|,\\&\leq\|\delta\Gamma_{AB}\|_2+|\delta\gamma|.\end{aligned}\tag{B.50}$$

Using the same method as for the equality constraints, we bound the final term with,

$$\left\langle\tilde{\Gamma}_{AB},\rho_{AB}\right\rangle\leq\tilde{\gamma}+(\dim(AB)+1)\,\epsilon_{\text{safe}}.\tag{B.51}$$

Similarly for lower bounds of the form $\langle\Gamma_{AB},\rho_{AB}\rangle\geq\gamma$, we get

$$\left\langle\tilde{\Gamma}_{AB},\rho_{AB}\right\rangle\geq\tilde{\gamma}-(\dim(AB)+1)\,\epsilon_{\text{safe}}.\tag{B.52}$$

- **Vector 1-norm:** We have the ideal version:

$$\begin{aligned}&\left\|\sum_{i=1}^{n}\langle\Gamma_i,\rho_{AB}\rangle\,\vec{e}_i-\vec{v}\right\|_1\leq\mu\\\Leftrightarrow&\left\|\sum_{i=1}^{n}\langle\Gamma_i,\rho_{AB}\rangle\,\vec{e}_i-\vec{v}\right\|_1-\mu\leq0.\end{aligned}\tag{B.53}$$

We swap each term to its finite representation counterpart and bound them as follows,

$$\begin{aligned}\left\|\sum_{i=1}^{n}\left\langle\tilde{\Gamma}_i,\rho_{AB}\right\rangle\vec{e}_i-\tilde{\vec{v}}\right\|_1-\tilde{\mu}&\leq\left\|\sum_{i=1}^{n}\langle\delta\Gamma_i,\rho_{AB}\rangle\,\vec{e}_i-\delta\vec{v}\right\|_1-\delta\mu+\left\|\sum_{i=1}^{n}\langle\Gamma_i,\rho_{AB}\rangle\,\vec{e}_i-\vec{v}\right\|_1-\mu\\&\leq\left\|\sum_{i=1}^{n}\langle\delta\Gamma_i,\rho_{AB}\rangle\,\vec{e}_i-\delta\vec{v}\right\|_1+|\delta\mu|\\&=\sum_{i=1}^{n}|\langle\delta\Gamma_i,\rho_{AB}\rangle-\delta v_i|+|\delta\mu|\\&\leq\sum_{i=1}^{n}(\|\delta\Gamma_i\|_2+|\delta v_i|)+|\delta\mu|\end{aligned}\tag{B.54}$$

If we assume that all of the scalar errors are around ϵ_{safe}, like with the previous constraints, we get a final bound of,

$$\left\|\sum_{i=1}^{n}\left\langle\tilde{\Gamma}_i,\rho_{AB}\right\rangle\vec{e}_i-\tilde{\vec{v}}\right\|_1\leq\tilde{\mu}+(n\,(\dim(AB)+1)+1)\,\epsilon_{\text{safe}}.\tag{B.55}$$

- **Matrix 1-norm:** We have the ideal version:

$$\begin{aligned}&\|\Phi_{AB\to C}(\rho_{AB})-\sigma_C\|_1\leq\mu\\\Leftrightarrow&\|\Phi_{AB\to C}(\rho_{AB})-\sigma_C\|_1-\mu\leq0.\end{aligned}\tag{B.56}$$

In the constrain class `MatrixOneNormConstraint` we store the Hermitian map in its Choi representation,

$$J_{\Phi}^{ABC} = \mathrm{id}_{AB} \otimes \Phi_{\tilde{A}\tilde{B}\to C}(\phi^{+}_{AB\tilde{A}\tilde{B}}), \tag{B.57}$$

where $\phi^{+}_{AB\tilde{A}\tilde{B}}$ is the canonical unnormalized maximally entangled state between systems AB and purifying systems $\tilde{A}\tilde{B}$ of the same dimension. Using the Choi matrix, we can rewrite the constraint as,

$$\left\|\mathrm{Tr}_{AB}[J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \sigma_C\right\|_1 - \mu \leq 0 \tag{B.58}$$

We swap each term to its finite representation counterpart and bound them as follows,

$$\begin{aligned}
&\left\|\mathrm{Tr}_{AB}[\tilde{J}_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \tilde{\sigma}_C\right\|_1 - \tilde{\mu} \\
&\leq \left\|\mathrm{Tr}_{AB}[J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \sigma_C\right\|_1 - \mu + \left\|\mathrm{Tr}_{AB}[\delta J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \delta\sigma_C\right\|_1 - \delta\mu \\
&\leq \left\|\mathrm{Tr}_{AB}[\delta J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \delta\sigma_C\right\|_1 - \delta\mu \\
&\leq \left\|\mathrm{Tr}_{AB}[\delta J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)]\right\|_1 + \|\delta\sigma_C\|_1 + |\delta\mu| \\
&\leq \left\|\delta J_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)\right\|_1 + \|\delta\sigma_C\|_1 + |\delta\mu| \\
&\leq \left\|\delta J_{\Phi}^{ABC}\right\|_2 \left\|(\rho_{AB}^{\mathrm{T}} \otimes I_C)\right\|_2 + \|\delta\sigma_C\|_1 + |\delta\mu| \\
&\leq \sqrt{\dim(C)}\left\|\delta J_{\Phi}^{ABC}\right\|_2 + \|\delta\sigma_C\|_1 + |\delta\mu|.
\end{aligned} \tag{B.59}$$

Like with the previous constraints we assume that all of the scalar errors are around ϵ_{safe}. We get a final bound of,

$$\left\|\mathrm{Tr}_{AB}[\tilde{J}_{\Phi}^{ABC}(\rho_{AB}^{\mathrm{T}} \otimes I_C)] - \tilde{\sigma}_C\right\|_1 \leq \tilde{\mu} + \left(\sqrt{\dim(C)}\dim(ABC) + \dim(C) + 1\right)\epsilon_{\mathrm{safe}}. \tag{B.60}$$

Although the bounds are more precise than before, their current form will still typically overestimate the required ϵ_{rep} for many constraints. For example, a `MatrixOneNormConstraint` that implements the partial trace over system B, $\Phi_{AB\to A} := \mathrm{id}_A \otimes \mathrm{Tr}_B$, would store it with the Choi matrix $\phi^{+}_{A\tilde{A}} \otimes I_B$. This Choi matrix is perfectly representable in memory with $\delta J_{\Phi}^{\tilde{A}BA} = 0$, but our method would choose an $\epsilon_{\mathrm{rep}} = \left(\dim(A)^{5/2}\dim(B) + \dim(A) + 1\right)\epsilon_{\mathrm{safe}}$. In addition to this, further knowledge of each constraint, such as block diagonal structure or relative magnitude, could be used to further reduce ϵ_{rep} for each constraint.

B.4.2 Careful Treatment of Perturbation

In [1, Theorem 2], Winick et al. provide a method to extend Eqs. (B.22) and (B.23) to a lower bound on the key rate when $\mathcal{G}(\rho)$ and/or $\mathcal{Z} \circ \mathcal{G}(\rho)$ have an eigenvalue of 0. This is especially important for Eq. (B.23) because the logarithm is not defined at 0. To work around this, Winick et al. perturb the state after applying the $\mathcal{G}$ and $\mathcal{Z}$ maps to remove non-positive eigenvalues. They choose the perturbation map

$$D_{\mathrm{aff},\epsilon}(\rho_{\bar{B}}) = (1-\epsilon)\rho_{\bar{B}} + \epsilon\frac{I_{\bar{B}}}{\dim(\bar{B})}, \tag{B.61}$$

and obtain

$$f_{\epsilon}(\rho_{\bar{A}}) := \mathrm{D}\left(\mathcal{G}_{\epsilon}\left(\rho_{\bar{A}}\right) \| \mathcal{Z} \circ \mathcal{G}_{\epsilon}\left(\rho_{\bar{A}}\right)\right), \tag{B.62}$$

and

$$\nabla f_\epsilon(\rho_{\bar{A}}) := \mathcal{G}_\epsilon^\dagger \left(\log \circ \mathcal{G}_\epsilon(\rho_{\bar{A}}) - \log \circ \mathcal{Z} \circ \mathcal{G}_\epsilon(\rho_{\bar{A}})\right), \tag{B.63}$$

with $\mathcal{G}_\epsilon := D_{\text{aff},\epsilon} \circ \mathcal{G}$. For simplicity, we are labelling all input systems to $\mathcal{G}$ as $\bar{A}$ and output systems as $\bar{B}$. Note that the map $D_{\text{aff},\epsilon}$ is slightly problematic as $D_{\text{aff},\epsilon}$ is not a linear map. Therefore, the dual map $D^\dagger_{\text{aff},\epsilon}$ is not defined. This did not affect their results because for a density matrix ρ, $D_{\text{aff},\epsilon}(\rho) = D_\epsilon(\rho)$, which we define below. At which point, one can substitute the incorrect, $D^\dagger_{\text{aff},\epsilon}$ with $D^\dagger_\epsilon$ in their proofs. We corrected and extend their proof by substituting $D_{\text{aff},\epsilon}$ with the true linear CPTP map,

$$D_\epsilon(\rho_{\bar{B}}) := (1-\epsilon)\rho_{\bar{B}} + \epsilon \operatorname{Tr}[\rho_{\bar{B}}] \frac{I_{\bar{B}}}{\dim(\bar{B})}. \tag{B.64}$$

We also improve upon their penalty term

$$\zeta_\epsilon := 2\epsilon(\dim(\bar{B}) - 1) \log\left(\frac{\dim(\bar{B})}{\epsilon(\dim(\bar{B}) - 1)}\right), \tag{B.65}$$

with

$$\tilde{\zeta}_\epsilon := 4\epsilon \frac{\dim(\bar{B}) - 1}{\dim(\bar{B})} \log\left(\frac{\dim(\bar{B})^2}{2\epsilon(\dim(\bar{B}) - 1)}\right). \tag{B.66}$$

and incorporate it into a new version of their Theorem 2. Furthermore, we determine a method for choosing a significantly smaller value ϵ to perturb by.

Extension of Previous Lemmas

In this section we rework and extend many of the lemmas from Ref. [1] for subnormalized states used in their Theorem 2. We also derive a smaller penalty term $\tilde{\zeta}_\epsilon$ compared to the original work's ζ_ϵ.

Define α_ϵ as

$$\alpha_\epsilon := \min_{\rho \in S \cap \operatorname{Pos}(\bar{A})} f_\epsilon(\rho), \tag{B.67}$$

and $\beta_\epsilon(\rho^{(0)}_{\bar{A}})$ as

$$\begin{aligned} \beta_\epsilon(\rho^{(0)}_{\bar{A}}) :=& f_\epsilon(\rho^{(0)}_{\bar{A}}) - \left\langle \rho^{(0)}_{\bar{A}}, \nabla f_\epsilon(\rho^{(0)}_{\bar{A}})\right\rangle + \text{dual problem}\left[\min_{\rho \in S \cap \operatorname{Pos}(\bar{A})} \left\langle \rho, \nabla f_\epsilon(\rho^{(0)}_{\bar{A}})\right\rangle\right], \\ =& \text{dual problem}\left[\min_{\rho \in S \cap \operatorname{Pos}(\bar{A})} \left\langle \rho, \nabla f_\epsilon(\rho^{(0)}_{\bar{A}})\right\rangle\right], \end{aligned} \tag{B.68}$$

where $\rho^{(0)}_{\bar{A}}$ is the chosen point to linearize at, taken from step 1. Our new version of Theorem 2 is,

Theorem 4 (Theorem 2 Extension for Positive Definite Matrices)**.** *Let* $\rho^{(0)}_{\bar{A}} \in \operatorname{Pos}(\bar{A})$ *such that* $\operatorname{Tr}\left[\rho^{(0)}_{\bar{A}}\right] \leq 1$ *and* $\operatorname{Tr}\left[\mathcal{G}(\rho^{(0)}_{\bar{A}})\right] > 0$*. Also, let* $\epsilon \in \left(0, \frac{\dim(\bar{B})}{2e(\dim(\bar{B})-1)}\right]$*, then*

$$\alpha_0 \geq \beta_\epsilon(\rho^{(0)}_{\bar{A}}) - \tilde{\zeta}_\epsilon. \tag{B.69}$$

The proof of Theorem 4 follows the same processes as [1, Theorem 2] but we substitute our extended versions of their lemmas in each step. Ultimately, Theorem 4 follows from our Lemma 10. We strongly encourage the reader to first read Appendix C of [1] before reading this section.

Before we begin, we rework their Lemma 1 to give sufficient conditions for the existence of our gradient.

Lemma 5 (Rework of [1, Lemma 1]: Existence of the Gradient)**.** *Let* $\rho \in \mathrm{Pos}(\bar{A})$ *such that* $\mathrm{Tr}[\mathcal{G}(\rho)] > 0$*, and let* $\epsilon \in (0, 1]$*. Then,*

$$\mathcal{G}_\epsilon(\rho) > 0, \tag{B.70}$$

$$\mathcal{Z} \circ \mathcal{G}_\epsilon(\rho) > 0, \tag{B.71}$$

and $\boldsymbol{\nabla} f_\epsilon(\rho)$ *exists.*

Proof. Note that $\mathcal{G}$ is a CPTNI map, $\mathcal{Z}$ a CPTP map, and $\mathcal{Z} \circ \mathcal{G}_\epsilon = D_\epsilon \circ \mathcal{Z} \circ \mathcal{G}$. For any $\rho \in \mathrm{Pos}(\bar{A})$ we get $\rho \geq 0$ and $\mathcal{Z} \circ \mathcal{G} \geq 0$. Now, for $\epsilon \in (0, 1]$, $\mathrm{Tr}[\mathcal{G}(\rho)] > 0$ ensures that $D_\epsilon \circ \mathcal{G}(\rho) > 0$ and $D_\epsilon \circ \mathcal{Z} \circ \mathcal{G}(\rho) > 0$. Furthermore, the logarithm in Eq. (B.63) now only acts on strictly positive definite operators. Therefore, the gradient exists at this point. □

A minor mistake was made in [1] when stating Fannes' inequality. Fannes' inequality requires the condition that ρ and σ must be *normalized* density matrices [2]. We use this to adapt Fannes' inequality for any two positive semidefinite matrices that have the same trace.

Lemma 6 (Adaptation of Fannes' Inequality)**.** *Let* $\rho, \sigma \in \mathrm{Pos}(A)$ *such that* $\mathrm{Tr}[\rho] = \mathrm{Tr}[\sigma]$*, and* $\|\rho - \sigma\|_1 \leq \kappa \leq \mathrm{Tr}[\rho] e^{-1}$*, then*

$$|H(\rho) - H(\sigma)| \leq -\kappa \log(\kappa) + \kappa \log(\mathrm{Tr}[\rho] \dim(A)). \tag{B.72}$$

Proof. Simply normalizes ρ and σ to have trace 1 and applies the original Fannes' inequality, then rearranges for the final form. □

Using our Lemma 6, we naturally extend [1, Lemma 6].

Lemma 7 (Extension of [1, Lemma 6]: Bounding the Relative Entropy.)**.** *Let* $\rho, \sigma \in \mathrm{Pos}(\bar{A})$ *such that* $\mathrm{Tr}[\mathcal{G}(\rho)] = \mathrm{Tr}[\mathcal{G}(\sigma)]$ *and* $\|\mathcal{G}(\rho) - \mathcal{G}(\sigma)\|_1 \leq \kappa \leq \mathrm{Tr}[\mathcal{G}(\rho)] e^{-1}$ *then*

$$|f(\rho) - f(\sigma)| \leq 2\kappa \left(-\log(\kappa) + \log\bigl(\mathrm{Tr}[\mathcal{G}(\rho)] \dim(\bar{B})\bigr) \right). \tag{B.73}$$

Unlike the previous lemmas our extension to [1, Lemma 7] also includes an improvement in the bound[6].

Lemma 8 (Extension and Improvement of [1, Lemma 7]: Bounding the Perturbation's Displacement)**.** *Let* $\rho \in \mathrm{Pos}(A)$ *and* $\epsilon \in [0, 1]$*, then*

$$\|(\mathrm{id}_A - D_\epsilon)(\rho)\|_1 \leq 2\epsilon \, \mathrm{Tr}[\rho] \frac{\dim(A) - 1}{\dim(A)}. \tag{B.74}$$

[6] We thank Shlok Nahar for finding this improvement.

Proof. We start by simplifying the left-hand side to get

$$\begin{aligned}\|(\mathrm{id}_A - D_\epsilon)(\rho)\|_1 &= \epsilon \left\| \rho - \mathrm{Tr}[\rho]\frac{I}{\dim(A)} \right\|_1 \\ &= \epsilon\,\mathrm{Tr}[\rho] \left\| \frac{\rho}{\mathrm{Tr}[\rho]} - \frac{I}{\dim(A)} \right\|_1 .\end{aligned} \tag{B.75}$$

Now, note for any normalized pure state $|\psi\rangle$ that

$$\begin{aligned}\left\| |\psi\rangle\langle\psi| - \frac{I}{\dim(A)} \right\|_1 &= 1 - \frac{1}{\dim(A)} + (\dim(A)-1)\frac{1}{\dim(A)} \\ &= 2\frac{\dim(A)-1}{\dim(A)}.\end{aligned} \tag{B.76}$$

Furthermore, all density matrices (including $\frac{\rho}{\mathrm{Tr}[\rho]}$) can be written as a convex combination of pure states, and using the convexity of the trace norm we get

$$\left\| \frac{\rho}{\mathrm{Tr}[\rho]} - \frac{I}{\dim(A)} \right\|_1 \leq 2\frac{\dim(A)-1}{\dim(A)}, \tag{B.77}$$

and the final statement

$$\|(\mathrm{id}_A - D_\epsilon)(\rho)\|_1 \leq 2\epsilon\,\mathrm{Tr}[\rho]\frac{\dim(A)-1}{\dim(A)}. \tag{B.78}$$

□

Using our extended and improved version of [1, Lemma 7] we extend and improve [1, Lemma 8] with our new $\tilde{\zeta}_\epsilon$.

Lemma 9 (Extension and Improvement of [1, Lemma 8]: Bounding the Relative Entropy after Perturbation)**.** *Let* $\rho \in \mathrm{Pos}(\bar{A})$ *such that* $\mathrm{Tr}[\mathcal{G}(\rho)] > 0$*, and let* $\epsilon \in [0, \frac{\dim(\bar{B})}{2e(\dim(\bar{B})-1)}]$*, then*

$$|f(\rho) - f_\epsilon(\rho)| \leq \mathrm{Tr}[\mathcal{G}(\rho)]\tilde{\zeta}_\epsilon. \tag{B.79}$$

Proof. The proof follows the same pattern as the original.

$$|f(\rho) - f_\epsilon(\rho)| \leq |H(\mathcal{G}(\rho)) - H(D_\epsilon \circ \mathcal{G}(\rho))| + |H(\mathcal{Z}\circ\mathcal{G}(\rho)) - H(D_\epsilon \circ \mathcal{Z} \circ \mathcal{G}(\rho))| \tag{B.80}$$

and

$$\|(\mathrm{id}_{\bar{B}} - D_\epsilon)(\mathcal{G}(\rho))\|_1 \leq 2\epsilon\,\mathrm{Tr}[\mathcal{G}(\rho)]\frac{\dim(\bar{B})-1}{\dim(\bar{B})} \leq \mathrm{Tr}[\mathcal{G}(\rho)]e^{-1}. \tag{B.81}$$

Similarly, it holds for $\|(\mathrm{id}_{\bar{B}} - D_\epsilon)(\mathcal{Z}\circ\mathcal{G}(\rho))\|_1$. We apply our Lemmas 6 and 8 with $\kappa = 2\epsilon\,\mathrm{Tr}[\mathcal{G}(\rho)]\frac{\dim(\bar{B})-1}{\dim(\bar{B})}$ and simplifies to,

$$|f(\rho) - f_\epsilon(\rho)| \leq \mathrm{Tr}[\mathcal{G}(\rho)]\tilde{\zeta}_\epsilon. \tag{B.82}$$

□

Lemma 10 (Extension of [1, Lemma 9]: Bounds on the Optimal Solution after Perturbation)**.** *Let* $\epsilon \in [0, \frac{\dim(\bar{B})}{2e(\dim(\bar{B})-1)}]$*. Let* $\rho_{min,0}$ *and* $\rho_{min,\epsilon}$ *be the optimal parameter choices for minimizing* α_0 *and* α_ϵ *respectively. If* $\mathrm{Tr}[\mathcal{G}(\rho_{min,0})] > 0$*, then*

$$\alpha_0 - \alpha_\epsilon \le \mathrm{Tr}[\mathcal{G}(\rho_{min,\epsilon})]\tilde{\zeta}_\epsilon \tag{B.83}$$

$$\alpha_0 - \alpha_\epsilon \ge -\mathrm{Tr}[\mathcal{G}(\rho_{min,0})]\tilde{\zeta}_\epsilon. \tag{B.84}$$

Proof. The proof is straight forward and simply makes use that (in general) $\rho_{\min,0}$ and $\rho_{\min,\epsilon}$ are not optimal solutions to each other's problems, then apply Lemma 9.

$$\begin{aligned} \alpha_0 - \alpha_\epsilon &= f(\rho_{\min,0}) - f_\epsilon(\rho_{\min,\epsilon}) \\ &\le f(\rho_{\min,\epsilon}) - f_\epsilon(\rho_{\min,\epsilon}) \\ &\le \mathrm{Tr}[\mathcal{G}(\rho_{\min,\epsilon})]\tilde{\zeta}_\epsilon, \end{aligned} \tag{B.85}$$

and,

$$\begin{aligned} \alpha_0 - \alpha_\epsilon &= f(\rho_{\min,0}) - f_\epsilon(\rho_{\min,\epsilon}) \\ &\ge f(\rho_{\min,0}) - f_\epsilon(\rho_{\min,0}) \\ &\ge -\mathrm{Tr}[\mathcal{G}(\rho_{\min,0})]\tilde{\zeta}_\epsilon. \end{aligned} \tag{B.86}$$

□

For Theorem 4, we do not know the value of $\rho_{\min,0}$[7], so we often use the simple bound $\mathrm{Tr}[\mathcal{G}(\rho_{\min,0})] \le 1$ from the restriction $\mathrm{Tr}[\rho] \le 1$ from Theorem 4. This gives,

$$\alpha_0 \ge \alpha_\epsilon - \tilde{\zeta}_\epsilon. \tag{B.87}$$

At this point we follow the same process as in [1] to show $\alpha_\epsilon \ge \beta_\epsilon(\rho_{\bar{A}}^{(0)})$ for our chosen linearization point. Combining with the above equation gives,

$$\alpha_0 \ge \beta_\epsilon(\rho_{\bar{A}}^{(0)}) - \tilde{\zeta}_\epsilon, \tag{B.88}$$

which finishes the proof of Theorem 4.

Logarithm Cutoff

Up to this point, we have improved [1, Theorem 2] into our Theorem 4 and reduced the penalty term ζ_ϵ to $\tilde{\zeta}_\epsilon$ (Eqs. (B.65) and (B.66)). The next set of major improvements comes from smarter choices of the perturbation term ϵ, and how to implement them into our numeric.

Both improvements are done in the context of numerical logarithms. At extreme values, the numerical stability of the matrix logarithm is significantly diminished. To give us some control over the matrix logarithm near 0, we implement the function, `logmsafe`, which extends,

$$\log_{\text{safe}}(x) = \log(\max\{x, \texttt{safeCutOff}\}), \tag{B.89}$$

for Hermitian matrices. We choose 0 < `safeCutOff` << 1 at an appropriate value to control our cutoff. Numerically, we implement this by decomposing into eigenvalues and eigenvectors, applying $\log_{\text{safe}}$ to the eigenvalues, then compiling back into a single matrix. Building on our `logmsafe` function, our numerical improvements are,

[7] If we did, then we wouldn't have to employ the 2 step solver in the first place.

- **Max Eigenvalue**: In theory, any value $\epsilon \geq 0$ should perturb the problem enough to ensure the gradient in Eq. (B.23) exists. However, numerical implementations of the logarithm in `logm` and `logmsafe` are stable only when the eigenvalues are distributed over a small range. Previously, the solution was to set a fixed minimum value to perturb by. However, for protocols with naturally small eigenvalues for $\mathcal{G}(\rho)$ and $\mathcal{Z} \circ \mathcal{G}(\rho)$, this would massively over-perturb the problem. Additionally, $\tilde{\zeta}_\epsilon$ would increase in magnitude as the eigenvalues of $\mathcal{G}(\rho)$ and $\mathcal{Z} \circ \mathcal{G}(\rho)$ decreased in magnitude.

  To overcome this, we select the cutoff by determining the largest eigenvalue of $\mathcal{G}(\rho)$ and $\mathcal{Z} \circ \mathcal{G}(\rho)$, $\lambda_{\max}$, and choosing a value of ϵ such that the minimum eigenvalue after perturbation is within a fixed fraction of $\lambda_{\max}$. From some simple testing we determined that setting the cutoff to ensure that the new $\lambda_{\min} \geq 10^{-14}\lambda_{\max}$ is the smallest we can make it while ensuring stability.

- **Smart Numerical Bounding**: As described above, all eigenvalues of $\mathcal{G}(\rho)$ and $\mathcal{Z} \circ \mathcal{G}(\rho)$ should be non-negative, but due to numerical instabilities, these values can sometimes be negative. Previously, this was solved by perturbing the eigenvalues enough such that all negative eigenvalues would end up larger than `logmsafe`'s eigenvalue cutoff.

  As pointed out by Shlok Nahar, this situation is purely the result of numerics and at worst these negative eigenvalues should actually be 0. Therefore, when we calculate our perturbation value ϵ we can treat negative eigenvalues as 0. Negative eigenvalues will not be perturbed above our safe cutoff of `safeCutOff` $= 10^{-14}\lambda_{\max}$, but a protection built into `logmsafe` will replace eigenvalues smaller than `safeCutOff` with `safeCutOff` anyway.