CONTENTS

Software Platform for Numerical Key Rate Calculation of General Quantum Key Distribution Protocols: User Guide

(Dated: September 3, 2021)

In this document, we provide a detailed user guide on using the program (in Sec. I), an introduction to the program architecture (in Sec. II) and guidelines for adding new protocols and solvers and keeping the interface consistent (in Sec. III). We provide a list of currently available solvers and protocols in Sec. IV.

For users interested in reading further, a more detailed documentation can be found in the Appendices. In Appendix A, we list all data structures, interfaces, and meaning of all parameters/options. Details on each solver, protocol, and optimization algorithm can also be found in the Appendices B-D.

I. USING THE PROGRAM

A. Installation

To install, please simply unzip the program zip package downloaded from the GitHub repository to a folder.

For software prerequisites, the program depends on the MATLAB and CVX libraries to run. Note that the versions MATLAB 2018b (or above) and CVX2 are required, while earlier MATLAB versions (which do not support string arrays and also some function syntaxes used) and CVX3 beta version are not compatible.

Optionally, the Mosek semidefinite programming solver can be installed, as an alternative solver to CVX's built-in SDPT3 solver (which is generally slower than Mosek). For decoy-state QKD protocols, the linear programming library Gurobi can be used in the channel model file (which will be faster than the default SDP solvers). Please refer to the respective documentations for CVX, Mosek, and Gurobi for their installation instructions.

When running the program, the MATLAB path should include the library folders (such as CVX, Mosek), the program root folder and all its subfolders. If one chooses to manually add the path, please make sure that only **one copy of this program exist on the search path** (as same-name functions of e.g. different history versions of the program can cause errors in the program).

Alternatively, one can uncomment and run the included automatic set path script in the beginning of the "main.m" file, which is first called to refresh the path before every single run:

```
restoredefaultpath;
addpath(genpath('.')) %add current software directory and all subfolders
%Windows path example
addpath(genpath('C:\cvx2')) %cvx
addpath(genpath('C:\Program Files\Mosek\9.0\toolbox\R2015a')) %external mosek
% %Mac/Linux path example
% addpath(genpath('~/cvx'))
% addpath(genpath('~/mosek'))
```

The lines `resetdefaultpath` and `addpath` here clears out all pre-exisiting paths in MATLAB and adds the current path (and its subfolders) as well as all required libraries. **Note that the script resets other path dependencies of MATLAB (such as other libraries), so please use it with caution if you have multiple libraries installed.**

At this point it is safe to simply run "Main.m" and begin using the program. If one would like to verify that CVX is set up correctly, though, one can run `cvx_setup` in the command line after the above code has been run at least once (such that CVX path is included in MATLAB) or after manually adding CVX to the MATLAB path. The command should output a list of available SDP solvers, which signifies that the path setup is successful.

B. Up and Running with a Simple Example

Once the necessary packages are installed and the paths are correctly included in MATLAB, please simply run "Main.m" to begin using the program. By default, "Main.m" is set to run a simple example for BB84, with a preset called `'pmBB84_asymptotic'` for BB84 performed over a lossy and misaligned channel, with infinite data size:

```
preset='pmBB84_asymptotic';
```

The next lines (which doesn't need modification) simply parse the preset file, and feed it to the main iteration function `mainIteration` to run and calculate key rate (which iterates over ranges specified in the preset, such as the distance to be scanned over, or an intensity that can be optimized, and calls the backend solver module to get upper/lower bounds on the key rate).

```
[protocolDescription,channelModel,leakageEC,parameters,solverOptions]=feval(
    preset);
results=mainIteration(protocolDescription,channelModel,leakageEC,parameters,
    solverOptions);
save('output.mat','results','parameters');
```

The main iteration results a struct `results`, which contains the scanned parameter (X-axis), the upper/lower bounds of the key rate, and other debugging information, which is saved to a ".mat" file. Additionally, the following output can be found in the command line window, which by default outputs the iteration number, Frank-Wolfe iteration results in the backend solver module, and the resulting upper/lower bound and running time. (It is also possible to reduce the verbose level in the preset file).

```
main iteration: 1
calculated closest rho
FW iteration:1    FW iteration time: 0.788898
projection value:0.346574    gap:0.000000    fvalue:0.346574    fgap:0.000000
upperBound: 0.453140, lowerBound: 0.453139
iteration time: 3.598005 s
```

If one would additionally like to plot the results (which is only useful if one iterates over multiple points), one can first change the last line of "Main.m" to for instance

```
plotResults(results,parameters,'km-log')
```

Then, one can edit the preset file `pmBB84_asymptotic` (which can be found in the "/Presets" folder. Alternatively, if "Main.m" has already been run once and the path has been set up, one can simply right click on the preset name in "Main.m" in `preset='pmBB84_asymptotic'`, and choose option "Open pmBB84_asymptotic" to view the preset file).

In the preset file "pmBB84_asymptotic", we can for instance change the parameters in function `setParameters`:

```
function parameters=setParameters(){
  parameters.names = ["ed","pz","pd","eta","etad","f","fullstat"];
  parameters.scan.eta = 10.^(-0.2*(0:5:20)/10); %channel transmittance
  parameters.fixed.ed = 0.01;
  parameters.fixed.pz = 0.5;
  parameters.fixed.pd = 0;
  parameters.fixed.etad = 1;
  parameters.fixed.fullstat = 1;
  parameters.fixed.f = 1.16;
}
```

such that `parameters.scan.eta` is changed from a single value to an array storing 5 values of transmittances from 0 km to 20 km. Note that there are other parameters used for the simulation too, which are fixed (i.e. unchanged at each iteration). The main iteration will be run for 5 times and a key rate is calculated for each `eta`.

We can now return to "Main.m" and run it. The main iteration will again be called (and it might take some time to finish all iterations), and plotting function should now show a plot of `eta` versus log10(Rate), as shown in Fig. 1 (a).

Alternatively, say if we want to plot the BB84 key rate over different levels of misalignment errors. We can then set in "Main.m" a linear plot:

```
plotResults(results,parameters,'linear')
```

And again modify the preset file "pmBB84_asymptotic.m":

```
function parameters=setParameters(){
  parameters.names = ["ed","pz","pd","eta","etad","f","fullstat"];
  parameters.scan.ed = 0:0.01:0.03; %misalignment
```

```
    parameters.fixed.eta = 1;
    parameters.fixed.pz = 0.5;
    parameters.fixed.pd = 0;
    parameters.fixed.etad = 1;
     parameters.fixed.fullstat = 1;
    parameters.fixed.f = 1.16;
  }
```

where we now set `ed` to be the variable that is scanned (from 0 to 0.03), and since we just want a fixed `eta` value and scan over `ed` instead, we set `parameters.fixed.eta=1`.

Now, again, we can return to "Main.m" and run it, which will return a plot of `ed` versus Rate, as shown in Fig. 1 (b).

Lastly, the user can obtain debugging information by uncommenting the following line in "Main.m"

```
results.debugInfo
```

which returns an array of `struct` that each stores the solver status, errors (and the exact line where they occurred) that MATLAB might have thrown, as well as names and values of parameters used in simulation, etc., at each sample point of the arrays specified in `parameters.scan`.

C. Using a Preset

As we have seen in the above example, to use the program, one can simply edit the entry-point file "Main.m" and choose a preset (which can either be an existing one or a custom one), and run "Main.m" to feed the preset into the solver and obtain the results. To choose a preset one simply specifies in "Main.m", such as:

```
preset='pmBB84_asymptotic';
```

We provide the following presets that combine selected protocols, solvers and corresponding parameters (the prefix is the protocol, and the suffix is the solver):

1. **pmBB84_asymptotic**: Prepare-and-measure BB84 with lossy & misaligned channel and asymptotic solver.
2. **pmBB84_finite**: Prepare-and-measure BB84 with lossy & misaligned channel and finite-size solver.
3. **pmBB84WCP_decoy**: Prepare-and-measure BB84 with weak coherent pulse (WCP) source and lossy & misaligned channel, and asymptotic solver with inequalities (for decoy-state analysis).
4. **MDIBB84_asymptotic**: Measurement-device-independent BB84 with single-photon source, lossy & misaligned channels and asymptotic solver.

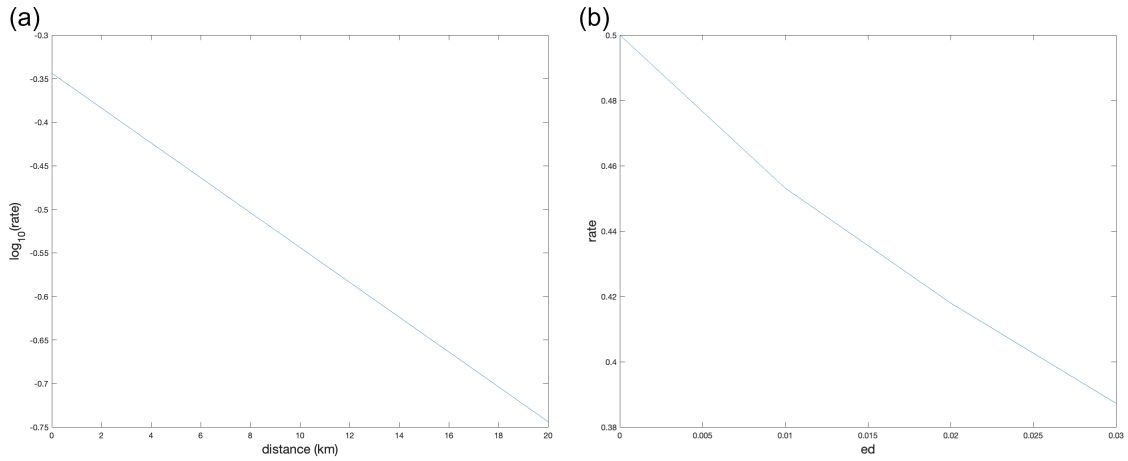

FIG. 1. Example simulation results from running the "pmBB84_asymptotic" example, for (a) key rate versus distance (choosing the "km-log" scale for X and Y axes) and (b) key rate versus misalignment error (choosing the linear scale for X and Y axes).

5. **MDIBB84_finite**: Measurement-device-independent BB84 with single-photon source, lossy & misaligned channels and finite-size solver.

6. **MDIBB84WCP_decoy**: Measurement-device-independent BB84 with WCP source, lossy & misaligned channels and asymptotic solver with inequalities (for decoy-state analysis).

7. **DMCVQKD_asymptotic**: Discrete-modulated continuous variable (CV) QKD (with heterodyne detection), with channel loss and excess noise, using asymptotic solver.

8. (archived) **pmBB84Simple_asymptotic**: Archived code for prepare-and-measure BB84 with a simple qubit model (no squashing model for detection) and an error channel with no loss, using asymptotic solver. This is the simplest protocol and can be used for testing if the program runs correctly.

9. (archived) **MDIBB84Simple_asymptotic**: Archived code for measurement-device-independent BB84 with single photon source and simple depolarizing channels, using asymptotic solver.

10. (archived) **MDIBB84Simple_asymptotic**: Archived code for measurement-device-independent BB84 with single photon source and simple depolarizing channels, using finite-size solver.

11. **templatePreset**: Custom template preset where the user can choose any description file and/or modify the parameters and settings.

One can modify the presets to set simulation parameters and solver options. Please refer to the next subsection I.D for details.

Once the preset is selected, one can simply run "Main.m" to generate key over specified parameters and the given descriptions. The output is recorded in an "output.mat" data file and (optionally) plotted on-screen with "plotResults" function.

D. Editing Presets

A preset file is stored in "/Presets" folder, and calls three functions (corresponding to protocol/channel description, parameters, and options) as listed below. For customization, one can edit a preset file so long as the three functions are provided:

- **setDescription**: choose the protocol, channel model, and error correction model to be used (which are external files with filenames "__Description", "__Channel", and "__EC", and are stored in "/Input" folder). The files are chosen by specifying corresponding function names. Note that one and only one of each of protocol/channel/error correction model should be specified. For error correction leakage, a protocol-independent file "generalEC" is provided. An example would be:

```
function [protocolDescription,channelModel,leakageEC]=setDescription()
  protocolDescription=str2func('pmBB84LossyDescription');
  channelModel=str2func('pmBB84LossyChannel');
  leakageEC=str2func('generalEC');
end
```

- **setParameters**: choose the protocol and channel parameters, such as basis choice probabilities, laser intensity, channel loss, misalignment, etc.. The parameters might affect the protocol description (e.g. basis choice probability affecting measurement operator), or might be used by the channel simulation (e.g. loss and misalignment in channel model).

  The parameters are listed under three types: `scan, fixed, optimize` (which form fields of a struct `parameter` to be returned to "Main.m"). The three types respectively stand for the parameter(s) to iterate over (every point in the array will be used to generate one instance of protocol for a key rate value), the parameters that stay unchanged, i.e. fixed, and the user-controllable parameters that can be optimized by searching (the array specifies a lower bound, the starting value, and the upper bound; and the software will try to sample multiple times to find the optimal value within the bounds that optimize key rate). Typical examples would be distance to iterate over (i.e. scan), misalignment to be fixed, and laser intensity to be optimized.

  There should be at least one "scan" parameter, and other two types are optional. Scannable parameters can be a single-point value or an array. Fixed parameters can be any type (values, arrays or matrices).

Optimizable parameters should be each in a three-element array of a given format of [lowerBound, startingValue, upperBound].

Additionally, a variable "names" (a string array), which specifies all parameter names used, should be included. The name list `names` should be a superset of the parameters used in the description files for the protocol, channel model, and error correction, whose input parameter requirements `varNames` are listed at the top of each file. Here we include the required parameter lists for our example protocol/channel combinations in Table I.

An example set of parameters would be:

```
1  function parameters=setParameters()
2
3    parameters.names = ["pz","ed","eta","etad","f","fullstat"];
4
5    parameters.scan.eta = 10.^(-0.2*(0:5:0)/10); %channel transmittance
6
7    parameters.fixed.pz = 0.5; %basis choice probability
8    parameters.fixed.ed = 0.01; %misalignment
9    parameters.fixed.etad = 1; %0.045; %detector efficiency
10   parameters.fixed.fullstat = 1; %whether to use full statistics
11   parameters.fixed.f = 1.16; %error correction efficiency
12
13   %parameters.optimize.pz = [0.1,0.5,0.9]; %basis choice probability
14
15 end
```

where we have set `eta` as the parameter to scan over (each point in the array will be evaluated), `ed,pz,pd,etad,fullstat,f` as the fixed parameters, and no optimizable parameter (optionally, in the commented out line 14, we can choose to e.g. optimize `pz` between 0.1 and 0.9, with 0.5 as the starting point if local search is used). Note that it is not mandatory to write parameters in `names` in the same order as appearance in the code as it will automatically be sorted.

- **setOptions**: choose the settings for the solver and the overall software platform. The most important parts are the names for solver 1 and solver 2 (which should be matching, such as `asymptotic/finite/asymptotic_inequality`), the max iteration number and maxgap of solver 1, the backend SDP library used, and the name and linear resolution for optimizer if there are optimizable parameters. It is also possible to turn on/off different levels of detail for the output messages by changing `globalSetting.verboseLevel` (for the entire program) and `optimizer.optimizerVerboseLevel` (for the optimizer, if used). One can simply use the existing list of options, and modify specific options of interest.

  An example set of options would look like (where we only list the key parameters):

```
1
2  function solverOptions=setOptions()
3
4    %key paremeters to be set
5    solverOptions.globalSetting.cvxSolver = 'sdpt3';
6    solverOptions.globalSetting.verboseLevel = 2;
7    solverOptions.solver1.name = 'asymptotic';
8    solverOptions.solver1.maxgap = 1e-6;
9    solverOptions.solver1.maxiter = 10;
10   solverOptions.solver1.removeLinearDependence = 'none';
11   solverOptions.solver2.name = 'asymptotic';
12
13   %key parameters for optimizer (if parameters.optimize is not empty)
14   solverOptions.optimizer.name = 'coordinateDescent';
15   solverOptions.optimizer.linearResolution = 5;
16   solverOptions.optimizer.iterativeDepth = 2;
17
18 end
```

In addition to just setting various parameters and solver options, for more advanced users, they can build upon an existing preset (or using the templatePreset) to combine arbitrary description/channel files and solvers, or even write their own descriptions/solvers, and link them using a custom preset. Such customization will be introduced in more detail in Section III.

E. Common Problems

Here we list some commonly met problems in setting up the program and potential solutions.

- Q: CVX solver is not found.

  A: Please try to run `cvx_setup` and check the returned list of available solvers. The `globalSetting.cvxSolver` field in the preset must match one of the available solvers. Also, if `cvx_setup` cannot be run, it might mean that the CVX installation path is not correctly found by MATLAB and might need to be manually added to MATLAB via the graphical interface or with `addpath(genpath('CVX_PATH'))`.

- Q: Mosek or Gurobi cannot be found by CVX, or they return empty values.

  A: Mosek and Gurobi are commercial solvers that require licenses. For academic users, both of them provide a free license option. Please obtain and activate the licenses for Mosek/Gurobi on their respective webpages, add their respective installation paths to MATLAB, and run `cvx_setup` again.

  Note that, CVX also includes an internal version of Mosek (if the user has another standalone Mosek installation path, one of them would appear as `mosek_2` in `cvx_setup`). To use the internal version, one can alternatively obtain an academic license from CVX, and rerun `cvx_setup`.

- Q: Running MATLAB only returns a function handle as output and not the simulation results.

  A: This might be the result of running from one of the preset or protocol files. Please always run the program from "Main.m" as the entry point to perform the calculation.

- Q: The solver runs but does not output the results/plot.

  A: Please check the option `verboseLevel` in the preset selected. Also, for plotting, please make sure that the desired plotting argument (among "linear", "linear-log", "km-log") is set in `plotResults` at the end of "Main.m", and it is not set to "none".

- Q: MATLAB returns an error about string array or "sum" function.

  A: The string array is introduced in MATLAB 2016b, and function syntax `sum(A,'all')` is introduced in MATLAB 2018b. Please kindly upgrade MATLAB to a more recent version and run the program again. If the problem only happens with the `sum` function, though, one can also circumvent it by replacing `sum(A,'all')` with `sum(sum(A,2))` in places where error is reported.

- Q: The first step solver returns a warning on non-successful status.

  A: CVX status other than success sometimes signal the existence of numerical difficulties (which could be due to theoretical or numerical reasons). Note that, for some cases, CVX might return a warning such as "failed" or "inaccurate" for some successfully solved QKD protocol models, for which the key rate is actually correct - if the upper and lower bounds both have non-zero values and their values are close, it usually signifies a successful solution.

- Q: The solver takes a long time, or returns inaccurate values.

  A: For most protocols, the two solver options (which can be set in the preset) that affect speed and accuracy the most are `maxiter` and `maxgap`. The user can try to set `verboseLevel` to higher, such as `3`, to see the gap obtained from each Frank-Wolfe iteration. If the gap is small and the function values is mostly unchanged across iterations, but the iteration does not stop, one can try increasing `maxgap` or decreasing `maxiter` to a hard limit such as 10 iterations (which is usually good enough to obtaining a tight bound). Additionally, `cvxSolver` affects the speed and accuracy too, and usually Mosek has a better performance than the default solver sdpt3 in CVX.

  For some protocols, it is normal for the solver to take more time due to larger matrix sizes to be optimized, such as MDI-QKD and DM-CV-QKD. Also, presets involving WCP sources and decoy-states will take more time to preprocess the channel data and perform decoy-state analysis as a linear program. This step will take some time (and will not be displayed in the main iteration outputs).

If the user includes a parameter in `parameters.optimize`, the search algorithm (such as `coordinateDescent` , a local search algorithm) will be activated, which will take substantially more time as the key rate is calculated multiple times to find the optimal `parameters.optimize` at each `parameters.scan` position. One can set a higher `optimizerVerboseLevel` to visualize the optimization progress, and also increase/decrease `linearResolution` (for brute force or local search) or `maxSteps` (for gradient descent), depending on whether the solution is under/over-optimized, to more efficiently allocate the compute time.

II. PROGRAM ARCHITECTURE

In this section we give a brief overview of the architecture of the software. The program is constructed by several modules, which are roughly categorized into user input, main iteration, and backend solvers. The software structure is illustrated in Fig. 2 (a), which shows the overall data flow. It contains:

- Input data: the description files (including protocol descriptions, channel and error correction model) and the user input that include parameters (which instantiate the protocol) and options (which specify the settings for the solver and parameter optimization algorithms). The description files provide a "blue-print" of protocols (with a parameter list waiting to be instantiated - such as channel loss, or misalignment), and the main iteration will generate an actual instance of a protocol/channel for each set of parameter values. The instantiation process is illustrated in Fig. 2 (b).
- Main iteration: an iteration that parses all input parameters, and iterates/optimizes over the search range of parameters. It generates an instance for the protocol for each set of parameter values, and calls the backend key rate calculation function for each instance. If the parameter optimization is enabled, the main iteration will try to generate key rate on multiple parameter sets and find the optimal set.
- Backend solver modules: the core part of the software that performs key rate calculation. It follow a two-step procedure [1] to lower-bound the key rate for a specific QKD protocol instance. It chooses the two backend solvers to connect to for steps 1 and 2, which are modularized and can be swapped in/out for different protocols. Also, backend solvers reduce the key rate bounding to a Semidefinite-Programming (SDP) problem, and need to be linked to an external SDP solver library to work.

The user can interact with the software on three abstraction levels: presets level (users simply pick a preset and modify the parameters and options to run), protocol level (user supplies the description and channel models to define a new protocol, and use an existing solver to calculate its key rate), and backend solver level (user creates a new solver module, which receive non-protocol-specific data and calculates key rate). An illustration of this architecture and its explanations can be found in Fig. 2 (b).

The description files return a *unified data structure* including observables and their expectation values, the kraus operators, the key maps, and the error-correction leakage (they're defined in the form of "blue-prints", which is a function of parameters, and they will be instantiated once a set of values are chosen for the parameters, as shown in Fig. 2 (c)). This data structure is largely independent of the protocol [1] and is passed to the key rate calculation module, which calls the solvers. In principle, a solver only cares about the set of data structure (which contains the observables and expecations which form POVM constraints, as well as the kraus operators and key maps that form the key rate function) and requires no knowledge of the protocol descriptions. Also, the main iteration only cares about the key rate value and the search range of parameters, and its parameter optimization process is independent of the protocol or solver, too. In this way the three main blocks are largely decoupled from each other, which faciliates addition of new protocols or solvers.

III. CUSTOMIZATION

To customize the program, please simply use the current description files or solver models as template, and follow the interface data structures/function formats.

---

[1] aside from occasional solver-specific inputs, such as total sent data size or security parameter required for finite-size solvers, which are not part of the description nor the channel, but are settings arbitrarily chosen by the user. Some protocols/solvers might also have a slightly modified key rate formula, in which case we might still require some knowledge of the parameters or the channel model, such as the proportion of single photons for protocols using WCP with decoy states.

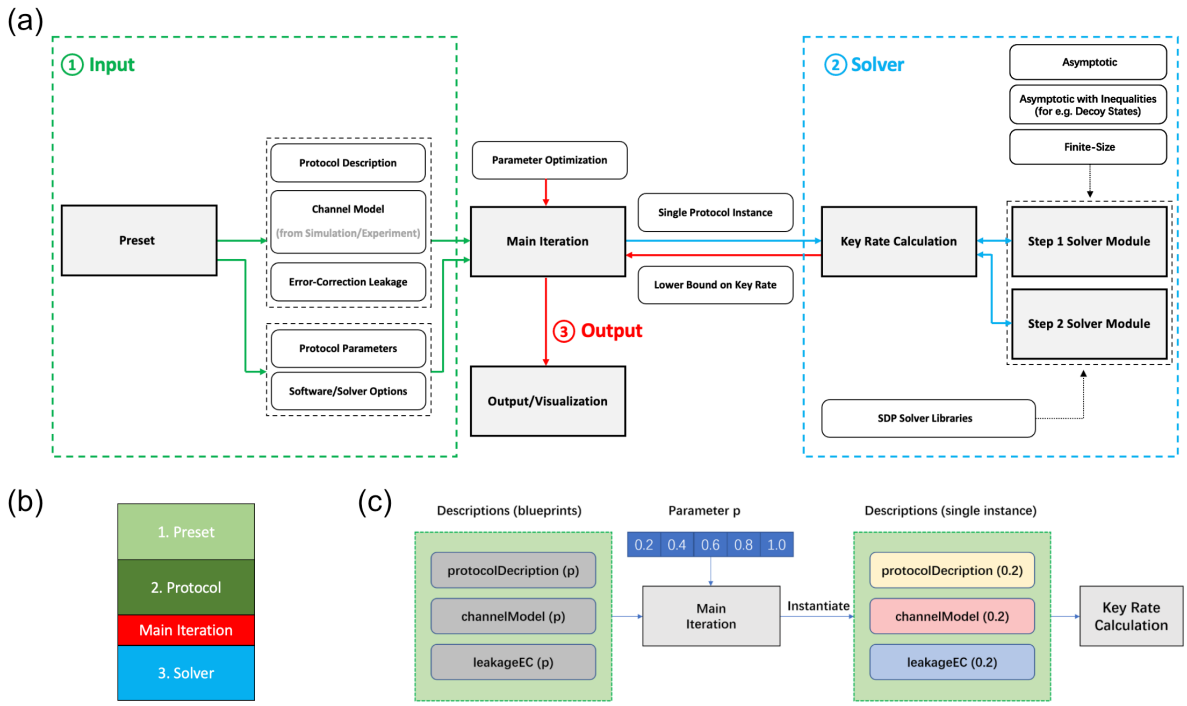

FIG. 2. (a) Illustration of the architecture of our software platform. 1. The user supplies a preset, which contains a set of descriptions for the QKD protocol, channel model, and error correction leakage, as well as the protocol parameters and solver options. 2. The main iteration calls the backend numerical solver each time for a single instance of the QKD protocol, corresponding to each set of parameters p. The solver performs a two-step numerical bounding of the secure key rate. 3. The main iteration scans (or optimizes) over the full range of input parameters, and outputs/visualizes the results. (b) The different abstraction levels of the software that the user can interact with. The preset specifies the protocol description and channel models (along with parameters and solver options), which is parsed in the main iteration and passed to the backend solvers to calculate the key rate. The user can interact with the program at 1. the preset level (simply choosing a preset to run, and modifying running parameters), 2. the protocol level (using existing solvers, and supplying new description/channel model files), and 3. the solver level (supplying a new backend solver module). (c) A more detailed look at the parameter instantiation process. The description files supplied by the user should be functions of a single set of parameter p. The main iteration reads the parameter list, and tests various samples of p. Each single set of p instantiates one instance of descriptions (which at this point become numerical arrays and matrices), which is fed into the solver to calculation one value of key rate.

A. Adding protocol

1. *Input*

If one would like to add a new protocol/channel model, one can simply use one of the existing description files as a template, and 1. modify function names for new protocols, 2. fill out the input parameter names `varNames` that are used in each file, 3. provide the necessary calculation process that uses the input parameters to derive the output data (`dimensions,krausOp,keyMap,observables` and optionally `obsMask` for protocol description, `expectations, pSift,errorRate/probDist` and optionally `expMask` for channel model, and `leakageEC` for error correction).

For input interfaces, all protocol description/channel model/error-correction files accept a set of parameter names `names` (declared in the preset), and parameter values `p` which contains actual values to the variables at run time, when mainIteration evaluates `p`. The description file should declare what parameters are used in the said file in a list `varNames` (which should be a subset of `names`).

The file always should start with the format of (taking a protocolDescription file as an example):

```
1  function protocolDescription = pmBB84LossyDescription(names,p)
2
3    varNames=["pz","fullstat"];
4
```

```
    %%%%%%%% interfacing (please do not modify) %%%%%%%%%%
    varValues = findVariables(varNames,names,p);
    addVariables(varNames,varValues);
    observables = {};
    obsMask = [];

    %%%%%%%% user-supplied description begin %%%%%%%%%%
```

where, importantly, `varNames` is the list of parameters used in this file and should be filled in. The interface checks `names` for matching parameter names and retrives their values in `p`. After these few interfacing lines, throughout the rest of the file these parameters declared in varNames (such as `pz, fullstat` here) can be used like any other MATLAB variable as if their values are known. The user can use them to calculate the necessary information such as `krausOp,keyMap,observables` for protocol description and `expectations` for channel model.

The input interfaces for channel model and error correction are largely the same. The only difference is that channel model also depends on an instance of `protocolDescription`, and error correction calculation depends on one instance of `channelModel`. So the start of the file for a channel model might look like:

```
function channelModel = pmBB84LossyChannel(protocolDescription,names,p)
  varNames=["ed","pz","eta","etad","fullstat"];
```

where it can additionally e.g. access the dimensions via:

```
    dimA = protocolDescription.dimensions(1);
    dimB = protocolDescription.dimensions(2);
```

For error correction, the start of the file looks like:

```
function leakageEC = generalEC(channelModel,names,p)
  varNames=["f"];
```

where it can additionally access channel model information such as:

```
    siftingFactor = channelModel.pSift;
    errorRate = channelModel.errorRate;
```

where we assume here the binary entropy is used for leakage calculation. (Alternatively `probDist` can be used, if it is recorded in `channelModel`).

2. Output

For output interfaces, the protocol description should at least return `krausOp,keyMap,observables` and `dimensions` (the dimensions of system for Alice and Bob).

Optionally, an `obsMask` can be included, which is a numerical array having the same length as `observables` and storing additional information on parsing the observables in specific solvers. For the asymptotic solver, no `obsMask` is needed. For the finite-size solver, the certain observables are marked with `obsMask=0` and the uncertain (with statistical fluctuation) observables are marked with `obsMask=1`. For the decoy_asymptotic solver, the certain observables are marked with `obsMask=0`, and the uncertain (with loosened bounds from decoy-state analysis) observables are marked with `obsMask=1`, while the corresponding uncertain expectations are marked with `expMask=1` and `expMask=2` respectively for lower and upper bounds acquired from decoy-state analysis.

The mask can be automatically generated if the user adds the observables using the function `addObservables`, such as in the following lines:

```
    addObservables(obs1);
    addObservables(obs2,'mask',1);
    addObservables({obs3;obs4},'mask',2);
```

which adds `obs1` to `observables` and generates a corresponding `obsMask` with the default mask value 0, adds `obs2` with a mask value 1, and adds multiple observables in one line (note that they should be matrices in a *single-column cell array*) with mask value 2 for each observable. If `addObservables` is used, the variables `observables` and `obsMask`

will be automatically generated and filled in. Of course, the user can still choose to fill in their own `observables` and (if needed) `obsMask` manually.

The end of a protocol description file should look like:

```
    %%%%%%%% user-supplied description end %%%%%%%%
    protocolDescription.krausOp = krausOp;
    protocolDescription.keyMap = keyMap;
    protocolDescription.observables = observables;
    protocolDescription.obsMask = obsMask;
    protocolDescription.dimensions = [dimA,dimB];
end
```

The channel model should at least return the expectations (and an optional expMask for specific solvers), as well as the information for error correction including pSift and either errorRate or probDist.

For error-correction, `pSift` is the overall probability of detection (including basis sifting factor) for the signal state, i.e. signal gain. If binary entropy is used, then `errorRate` should be the signal state QBER. Alternatively, `channelModel` can return the full probability distribution of Alice's input and Bob's output (and not just error rate) `probDist` for a better bound on the error-correction leakage. The error-correction model checks for the existence of these fields and takes the according model.

Similar to the observables, an `expMask` can be optionally included, which is a numerical array having the same length as `expectations` and storing additional information on parsing the expectations in specific solvers. For the asymptotic solver, it is not required. For the finite-size solver, certain observables have `expMask=0` and uncertain observables have `expMask=1`. For the asymptotic solver, certain observables have `expMask=0` and uncertain observables have `expMask=1` as their lower bound and `expMask=2` as their upper bound.

Again, the mask can be automatically generated if the user adds the expectations using the function `addExpectations` , such as in the following lines:

```
    addExpectations(exp1);
    addExpectations(exp2,'mask',1);
    addExpectations([exp3,exp4],'mask',2);
```

which either adds an expectation with a default `expMask=0`, with a specified mask value, or adds multiple expectations at once. Note that the difference here is that `addExpectations` accepts either numerical values (like `exp1` or `exp2`) or a single-column numerical array (like `[exp3;exp4]`) which is different from `addObservables` which accepts a matrix or a cell array of matrices.

The end of a channel model file should look like (assuming binary entropy model for error-correction):

```
    %%%%%%%% user-supplied channel model end end %%%%%%%%
    channelModel.expectations = expectations;
    channelModel.expMask = expMask;
    channelModel.errorRate = [errorx,errorz];
    channelModel.pSift = [gainx,gainz];
end
```

Lastly, the error-correction file is relatively straightforward (and the default "generalEC.m" is usually sufficient for most protocols and doesn't need modification), which returns a numerical value `leakageEC` that estimates error-correction leakage.

Once the protocol description or channel model is added, one can simply modify an existing preset (or write a new one based on "templatePreset.m") to fill in the rest of the parameters and options information, and use this preset in the main function.

B. Adding solver

If one would like to add a new solver, please add a new case in the switch clause in "getKeyRate.m" for step 1 and/or step 2, and also the key rate calculation (if a modified key rate formula is used).

The solver in "getKeyRate.m" would receive the data structures `protocolDescription` and `channelModel`, which contain the above mentioned information including `krausOp,keyMap,observables,expectations`. If needed, it can also access the arrays `obsMask` and `expMask` to parse observable/expectation into groups (e.g. inequality and equality constraints).

A solver can also optionally access the parameter set `p` (such as directly reading data size `N` or security parameter `eps`, which are parameters not directly involved in the protocol description and are not in the data structures `protocolDescription` and `channelModel`.

Overall, the interfaces for the description files and the solver are listed in Table II.

As an example, a step 1 solver in"getKeyRate.m" can retrieve information such as

```
dim = protocolDescription.dimensions;
krausOp = protocolDescription.krausOp;
keyMap = protocolDescription.keyMap;
observables = protocolDescription.observables;
obsMask=protocolDescription.obsMask;

expectations = channelModel.expectations;
expMask=channelModel.expMask;

options = solverOptions.solver1;

[rho, upperBound] = solver1(dim, observables, expectations, obsMask, expMask,
    krausOp, keyMap, options);
```

Additionally, it may not be possible to represent some parameters (such as data size `N` or security parameter `eps` for finite-size solver) by the four pieces of data `krausOp,keyMap,observables,expectations` alone. In these cases, the full parameter list `p` and the parameter names are also directly accessible to the solver, if needed, using the below methods:

```
%additional parameters that can be read from full input parameter list
N=findParameter("N",names,p);
ptest=findParameter("ptest",names,p);
eps=findParameter("eps",names,p);
```

after which these variables `N,ptest,eps` can be used to calculate additional information, before calling the solver.

A step 2 solver might look like:

```
options = solverOptions.solver2;
lowerBound = solver2(rho, dim, observables, expectations, obsMask, expMask,
    krausOp, keyMap, options);
```

which uses the same input information plus the density matrix `rho` acquired in step 1, and returns the lower bound on the key rate. The lower bound can then be passed to the final step in key rate calculation to calculate the privacy amplification required and combine with the error-correction leakage, to get the final key rate.

A list of input and output interfaces for descriptions and solvers can be found in Appendix A.

IV. LIST OF SOLVERS AND EXAMPLE PROTOCOLS

The current version of the software package includes mainly the contents of three papers, Refs. [1–3]. There are three types of protocol included (with accompanying channel models):

- *Standard BB84*: A prepare-and-measure scheme where Alice encodes her bit in four states from two bases, to be measured by receiver Bob. We include a version using single-photon source with a qubit squashing model for detection, as in Refs. [1, 2], and with a lossy & misaligned channel model, compatible with asymptotic and finite-size solvers. We also include a version with WCP source and decoy states, as in Ref. [3], compatible with asymptotic solver (decoy-state version).

- *Measurement-device-independent QKD*: A measurement-device-independent scheme where both Alice and Bob encodes bit in signals to send to a third-party Charlie. We include a version with single-photon source as in Ref. [1] compatible with asymptotic and finite-size solvers. We also include a version with WCP source and decoy states, as in Ref. [3], compatible with asymptotic solver (decoy-state version).

- *Discrete-Modulated Continuous Variable QKD (DM-CVQKD)*: A prepare-and-measure CVQKD scheme where Alice encodes in discrete-modulated coherent states, and Bob performs heterodyne measurements. The channel includes loss and excess noise. The description and channel model are based on Ref. [4]. This protocol description is compatible with the asymptotic solver.

and there are three solvers:

- *Asymptotic (infinite data size) solver* based on Ref. [1], using near-equality constraints (given a small numerical tolerance).
- *Asymptotic solver with inequalities* based on Ref. [3], which is a modified version of the asymptotic solver that can use inequality constraints, such as those resulting from decoy-state analysis, which estimates upper and lower bounds of statistics for each photon number states (which are not tight bounds due to the finite number of decoys).
- *Finite-size solver* based on [2]. There are inequality constraints using the "mu-ball" method in the paper to bound statistical fluctuations. Note that the finite-size solver currently doesn't work with decoy-state protocols.

For parameter optimization, there are a few optimizers to choose from:

- *Brute-force Search*: In this algorithm we can simply search over the entire parameter space for all possible components of $\vec{p}$, subject to a finite resolution. This is effective for smaller numbers of parameters and coarse resolutions, but the search time exponentially grows with the number of parameters.
- *Coordinate Descent (recommended)*: Algorithm used in Ref. [5, 6]. Each component (i.e. coordinate) of $\vec{p}$ is updated one at a time while keeping other components fixed. This is a type of local search similar to gradient descent, and its advantage is the possibility of parallelization (for the linear search) and its being less likely to get trapped locally by small numerical noise.
- *Gradient Descent*: A standard gradient descent method. A small sample step in each component direction is taken to evaluate the partial derivaties, which are then combined to form the gradient. The algorithm stops when maximum iteration number is reached or if two iterations return close enough values.
- *Adam Optimizer [7]*: An adaptive learning rate local search algorithm commonly used in machine learning fields (but can be used for parameter optimization of a convex function, too). Adam optimizer is considered a versatile local search algorithm that works well against difficult landscapes (i.e. function shapes) and converges faster than gradient descent.

More details on each of the solvers, protocols, and optimization algorithms can be found in the documentation in the appendix Sections B-D.

TABLE I. Required parameters for example protocol/channel pairs. The parameters marked in blue are simple Boolean software flags that turn some options on/off. The parameters are respectively misalignment e_d (or misalignment angles θ_A, θ_B for MDI-QKD), basis choice probability p_z, dark count probability p_d, channel loss η, detector efficiency η_d, error correction efficiency f, coarse/fine-grain data flag $fullstat$, decoy intensities $\{\mu_1, \mu_2, \mu_3\}$, active/passive detection model flag $active$, depolarizing probability dp, excess noise ξ, source amplitude α, amplitude and phase post-selection flags amp_ps, phase_ps, photon number cutoff N_{cutoff}, and reverse/direct reconciliation flag recon. Note that $f \geq 1$ for DVQKD protocols, and by convention, the error correction efficiency $\beta \leq 1$ for CVQKD. Here for convenience, if using CVQKD protocol, f means β (i.e. they use the same parameter slot, but one can simply fill in $f \leq 1$ for CVQKD protocols to represent β).

Protocol	Channel	Required Parameters
pmBB84SimpleDescription	pmBB84ErrorChannel	p_z, e_d, f
pmBB84LossyDescription	pmBB84LossyChannel	p_z, e_d, η, η_d, f, fullstat
pmBB84LossyDescription	pmBB84WCPChannel	p_z, e_d, η, η_d, p_d, $\{\mu_1, \mu_2, \mu_3\}$, f, active, fullstat
MDIBB84Description	MDIBB84DepolarizingChannel	p_z, dp, f
MDIBB84Description	MDIBB84LossyChannel	p_z, θ_A, θ_B, η, f
MDIBB84Description	MDIBB84WCPChannel	p_z, θ_A, θ_B, η, p_d, $\{\mu_1, \mu_2, \mu_3\}$, f
DMCVHeterodyneDescription	DMCVHeterodyneChannel	η,ξ,α,N_{cutoff},f,amp_ps,phase_ps,recon

TABLE II. Input and output interfaces for descriptions and solvers. Here "observations, expectations" are abbreviated as "obs., exp." due to lack of space. The optional fields are marked in blue. Arrays *obsMask,expMask* provide additional information on parsing observables/expectations into groups, and are optional for descriptions. Channel model not only generates expectations, but also *pSift* and *errorRate/probDist* (choose one to use - for binary entropy or the full distribution) used to calculate leakage. Note that the channel model requires one instance of *protocolDescription* to obtain dimensions, and the error correction calculation requires one instance of *channelModel* to obtain pSift and *pSift* and *errorRate/probDist*.

Module	Input interfaces	Output interfaces
Protocol Description	p, names, varNames	krausOp, keyMap, obs., obsMask, dimensions
Channel Model	protocolDescription, p, names, varNames	exp., expMask, pSift, errorRate/probDist
Error Correction	channelModel, p, names, varNames	leakageEC
Solver	krausOp, keyMap, obs., exp., obsMask, expMask, p	lowerBound, FWBound, upperBound

Appendix A: Documentation: Data Structure and Interfaces

In this section, we list tables of input/output interfaces of description files and solvers, as well as the data structures used in the presets.

Table I lists the required parameters used for each the example protocol/channel pairs included in the software package. These parameters should be provided in the preset in `setParameters`.

Table II lists the required and option input/output interfaces for the description files and the solvers, which one should follow if adding new description files or adding a new solver.

Table III lists in detail all the input and output data structures used for the protocol description/channel model/error-correction, as well as parameters and solver options defined in the preset. Detailed explanation for each variable can be found in the table.

Table IV lists all the solver options (which can be modified in the preset). The options marked in bold either are mandatory or have a big influence on the performance of the solvers. Not all options need to be declared explicitly, as not only the preset includes example values for all options, the solver also automatically performs checking and assumes default values for missing options.

Appendix B: Documentation: Solver Modules

In this section we will introduce the solver modules that follow a two-step process to bound the key rate, based on input data of protocol description (POVM observable operators, kraus operators, and key maps) and channel model (expected values of observables). This is only meant to be a brief overview of the solver modules, and more detailed descriptions can be found in Ref. [1–3], for asymptotic, finite-size, and decoy-state enabled solvers, respectively.

There are three example solvers currently provided with the package:

- *Asymptotic (infinite data size) solver* based on Ref. [1].

TABLE III. Input and output data structures used in the protocol description, channel model, error-correction, parameters, and solver options. Parameters, solverOptions, and numerical are abbreviated as para., opt., and num. due to lack of space.

Name	Required	Type	Note
p	Yes	Cell Array	full list of parameter values used as input to all description files; its actual values will be automatically instantialized during `mainIteration`; the range p can take is defined in the preset `setParameters` function.
names	Yes	String Array	full list of parameter names used as input to all description files; its values are defined in the preset `setParameters` function.
varNames	Yes	String Array	the list of parameters actually used in a description file; the user should declare it at the beginning of each description file.
krausOp	Yes	Cell Array	used to calculate $\mathcal{G}(\rho) = \sum_i K_i \rho K_i^\dagger$ in `primalf`
keyMap	Yes	Cell Array	used to calculate $\mathcal{Z}(\mathcal{G}(\rho)) = \sum_j Z_j(\mathcal{G}(\rho))Z_j$ in `primalf`
observables	Yes	Cell Array	each element corresponds to a matrix representation of the POVM.
dimensions	Yes	Num. Array	stores each party's system dimension in order, e.g. $[dim_A, dim_B]$.
obsMask	Opt.	Num.Array	same length as observables; stores flags used to parse observables in groups.
expectations	Yes	Num. Array	same length as observables; stores $\gamma_i = Tr(\rho\Gamma_i^\dagger)$ based on channel model.
expMask	Opt.	Num. Array	same length as expectations; stores flags used to parse expectations in groups.
pSift	Yes	Num. Array	probability of signal entering key generation (including sifting/loss); can be more than 1 set.
errorRate	Yes	Num. Array	error rate for "correct/incorrect" events used for leakage calculation; can be more than 1 set; if used, probDist should not be included.
probDist	Yes	Num. Array	full statistics for Alice/Bob used for leakage calculation; row for Alice and column for Bob; if used, errorRate should not be included.
leakageEC	Yes	Num. Value	calculated error-correction leakage (a single numerical value)
para.scan	Yes	Struct	at least one scannable parameter should be specified (can be length of one).
para.fixed	Opt.	Struct	each field can be any type including numerical values, arrays or matrices.
para.optimize	Opt.	Struct	each field should be specified in [lowerBound, startingValue, upperBound].
opt.globalSetting	Yes	Struct	specifies SDP solver library precision and verbose option.
opt.optimizer	Opt.	Struct	should choose an optimizer name if p.optimize is not empty.
opt.solver1	Yes	Struct	must choose a step 1 solver module name.
opt.solver2	Yes	Struct	must choose a step 2 solver module name.

- *Asymptotic solver with inequalities* based on Ref. [3].
- *Finite-size solver* based on Ref. [2].

Here we first generally describe the goal of a numerical solver (which is common for all solvers). Using the Renner framework [13] and the formulations in Refs. [1, 9], the key rate can be lower-bounded as:

$$R = min_{\rho \in S} f(\rho) - p_{pass} \times \text{leak}_{obs}^{EC} \tag{B1}$$

where ρ is all possible density matrices Eve can input into Alice and Bob's systems while still keeping consistency with physical observables (which serve as constraints for ρ). $leak_{obs}^{EC}$ is the leakage caused by performing error-correction.

As described in [1], the privacy amplification (former term) is effectively an optimization problem of finding ρ that minimizes $f(\rho)$, subject to the constraints given by observables $\{\Gamma_i\}$ and their expected values $\{\gamma_i\}$.

The function $f(\rho)$ takes the form of quantum relative entropy:

$$f(\rho) = D(\mathcal{G}(\rho)||\mathcal{Z}(\mathcal{G}(\rho))) \tag{B2}$$

where $D(X||Y) = Tr(XlogX) - Tr(XlogY)$. Here $\mathcal{G}$ and $\mathcal{Z}$ are respectively the maps representing the kraus operators and the key map.

TABLE IV. Detailed descriptions of solver options, which can be categorized into four structs: globalSetting, optimizer, solver1, and solver2. Most presets already have the fields filled out, so the user only needs to modify what is needed. The mandatory parameters and key parameters affecting performance are marked in bold. Also, by the default the solvers and optimizers check for missing fields, and will use default values if missing.

Category	Name	Example Value	Explanation
globalSetting	**cvxSolver**	'sdpt3'	name of the SDP solver library: 'sdpt3' or 'mosek'.
globalSetting	**cvxPrecision**	'high'	precision of the SDP solver library, typically 'high' or 'best'.
globalSetting	verboseLevel	2	the output level of the program between 0 to 3.
optimizer	name	'coordinateDescent'	name of the parameter optimizer, choose among 'coordinateDescent', 'bruteForceSearch', 'gradientDescent' and 'localSearch_Adam'.
optimizer	**maxIterations**	2	max rounds of iterations over all axes for coordinateDescent.
optimizer	linearSearchAlgorithm	'iterative'	linear search algorithm for coordinateDescent: 'iterative' or 'fminbnd'.
optimizer	iterativeDepth	2	max linear iteration depth for coordinateDescent if 'iterative' is selected.
optimizer	linearResolution	5	search resolution (sample points) along each axis for bruteForceSearch or coordinateDescent during linear search.
optimizer	maxSteps	10	max search steps for gradientDescent and localSearch_Adam.
optimizer	optimizerVerboseLevel	1	the output level of the optimzier between 0 to 2.
solver1	**name**	'asymptotic'	step 1 solver name: 'asymptotic','asymptotic_inequality','finite'.
solver1	**maxiter**	10	max Frank-Wolfe iterations (if maxgap condition not met).
solver1	**maxgap**	1e-6	stopping criteria for Frank-Wolfe iterations.
solver1	**initmethod**	1	initialization of `rho0` in `closestDensityMatrix`; 0: minimizes `norm(rho0-rho)`; 1: minimizes `-lambda_min(rho)`.
solver1	linearconstrainttolerance	1e-10	tolerance of constraints in SDP.
solver1	linearsearchprecision	1e-20	precision for the linear search `fminbnd`.
solver1	linearsearchminstep	1e-3	min step size for the linear search `fminbnd`.
solver1	maxgap_criteria	true	true/false: using gap or primalf value as stopping criteria.
solver1	removeLinearDependence	'none'	preprocessing of constraints: 'none', 'rref'.
solver2	**name**	'asymptotic'	step 2 solver name: 'asymptotic','asymptotic_inequality','finite'; should match step 1 solver name.
solver2	epsilon	0	reserved to check the amount of perturbation to `rho` (reserved field for debugging, not currently used).
solver2	epsilonprime	1e-12	amount of loosening for constraints (see Ref. [1] Eq. 21); may be optionally overrided in `getKeyRate` to be the max constraint violation obtained in step 1.

$$\begin{aligned} \mathcal{G}(\rho) &= \sum_i K_i \rho K_i^\dagger \\ \mathcal{Z}(\mathcal{G}(\rho)) &= \sum_j Z_j(\mathcal{G}(\rho)) Z_j \end{aligned} \tag{B3}$$

which means the key rate can be calculated for a specific ρ given the kraus operators and the key maps. The goal of the solvers, therefore, is to solve for the optimal density matrix ρ^* that minimizes $f(\rho)$.

1. Asymptotic Solver

The asymptotic solver is based on Ref. [1]. In the asymptotic case, the observables and expectations form equality constraints, and the problem is modeled as:

$$\begin{aligned} minimize \quad & f(\rho) \\ Tr(\Gamma_i \rho) &= \gamma_i \\ \rho &\in \mathbf{H}_+ \end{aligned} \tag{B4}$$

However, $f(\rho)$ is not a linear function, therefore this is not a directly solvable semidefinite-programming (SDP) model. To solve this problem, as described in Ref. [1], a two-step process is performed, based on the observation that $f(\rho)$ is a convex function.

In the first step, one performs a local search to find near-optimal ρ - here we use the Frank-Wolfe algorithm:

1. At each ρ_i (starting from ρ_0), calculate the gradient $\nabla f(\rho_i)$, and solve the SDP problem

$$\begin{aligned} variable \quad & \Delta\rho \\ minimize \quad & Tr((\Delta\rho)^T \nabla f(\rho_i)) \\ Tr(\Gamma_i(\rho_i + \Delta\rho)) &= \gamma_i \\ (\rho_i + \Delta\rho) &\in \mathbf{H}_+ \end{aligned} \tag{B5}$$

   which finds a step $\Delta\rho$ along the direction of descent that satisfies the constraint.

2. Then, a linear search is performed to find the best interpolation between ρ_i and $\rho_i + \Delta\rho$

$$\begin{aligned} \lambda^{opt} &= argmin(f(\rho_i + \lambda\Delta\rho)) \\ \rho_{i+1} &= \rho_i + \lambda^{opt}\Delta\rho \end{aligned} \tag{B6}$$

3. Steps 1 and 2 are iterated multiple times until iteration count reaches maximum or a sufficiently low gap difference $|Tr(\rho_{i+1}^T \nabla f(\rho_i)) - Tr(\rho_i^T \nabla f(\rho_i))|$ (or function value difference $|f(\rho_{i+1}) - f(\rho_i)|$) is found, which signifies near-optimality and can terminate the algorithm.

At the end of step 1, the solver finds a ρ that is close to the actual optimal value ρ^*. Generally speaking, step 1 is the main contributor to the computing time (due to the multiple iterations - but of course, this depends on how fast the Frank-Wolfe algorithm converges for the specific instance of the protocol).

For the second step, the SDP is converted to its dual problem

$$\begin{aligned} variable \quad & \vec{y} \\ maximize \quad & \vec{\gamma} \cdot \vec{y} \\ \sum_i y_i \Gamma_i^T &\leq \nabla f(\rho) \\ \vec{y} &\in \mathbf{R}^n \end{aligned} \tag{B7}$$

where $\vec{\gamma}$ is the array of expectations, and ρ is the near-optimal value found in step 1.

For Γ, γ here, in fact to simplify the formulation, it is assumed that $Tr(\Gamma_i \rho) \leq \gamma$ and $-Tr(\Gamma_i \rho) \leq -\gamma$, so the overall set of constraints can be written as $\{\Gamma_i, -\Gamma_i\}$ and $[\gamma_i, -\gamma_i]$ such that one only needs to consider the inequality constraint $Tr(\Gamma\rho) \leq$.

After obtaining $max\vec{\gamma} \cdot \vec{y}$, the lower bound on the optimal value of $f(\rho^*)$ can be obtained as

$$\beta(\rho) = f(\rho) - Tr(\rho^T \nabla f(\rho)) + max\vec{\gamma} \cdot \vec{y} \tag{B8}$$

which is a reliable numerical lower bound on the privacy amplification part of the key rate. One can then subtract $leakageEC$ to obtain the final key rate. The two step procedure can be illustrated in Fig. 3.

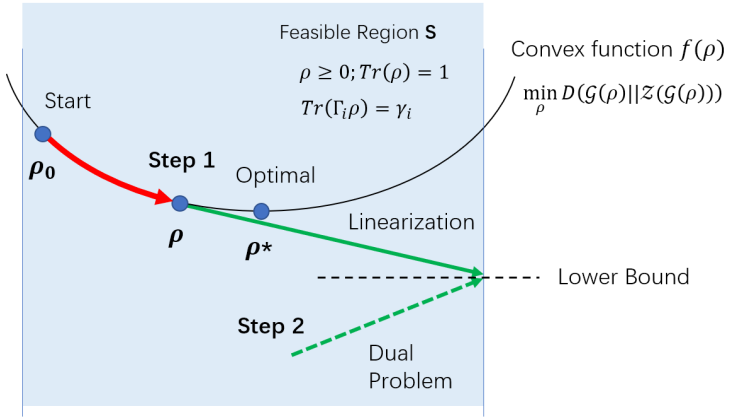

FIG. 3. Illustration of two-step procedure of the asymptotic solver to bound the key rate, reproduced from Ref. [1]. The problem is a minimization of a convex function $f(\rho)$ over a feasible region defined by constraints. The first step finds a near-optimal ρ by iterating with Frank-Wolfe algorithm (which are multiple SDP problems and linear searches), while the second step uses this ρ to perform a linearization and provide a reliable lower bound on the function value (by solving a dual problem).

2. Asymptotic Solver with Inequalities

In the case of protocols with decoy-states, for some observables Γ_j the expectation values γ_j from single photon components of the sources cannot be determined exactly, but rather follows a pair of lower and upper bound γ_j^L, γ_j^U (which can be derived from the decoy-state analysis). Meanwhile some observables Γ_i can still be known for certain (e.g. characterization of Alice's source) and do not need decoy states. The step 1 SDP problem simply becomes [3]:

$$\begin{aligned} variable \quad & \Delta\rho \\ minimize \quad & Tr((\Delta\rho)^T \nabla f(\rho_i)) \\ & \gamma_j^L \leq Tr(\Gamma_j(\rho_j + \Delta\rho)) \leq \gamma_j^U \\ & \gamma_i \leq Tr(\Gamma_i(\rho_i + \Delta\rho)) \leq \gamma_i \\ & (\rho_i + \Delta\rho) \in \mathbf{H}_+ \end{aligned} \tag{B9}$$

For step 2, as mentioned above, the solver already caters for inequality constraints. For observables we know for certain (e.g. characterization of source) we can still use $\{\Gamma_i, -\Gamma_i\}$ and $[\gamma_i, -\gamma_i]$, while for uncertain observables we can instead simply use $\{\Gamma_j, -\Gamma_j\}$ and $[\gamma_j^U, -\gamma_j^L]$ and call the step 2 solver as usual.

3. Finite-size Solver

The finite-size solver is based on Ref. [2]. The basic two-step procedure is similar to the asymptotic case, but here the **expectations** data actually represent the experimentally observed **frequencies** F, which contain statistical fluctuations and can deviate from the actual expected values. The fluctuations for all POVMs are bound by a "μ-ball" that satisfies

$$\begin{aligned} \mu_{coherent} &= 2\sqrt{\frac{1}{m}(log_2(1/\epsilon_1) + \Sigma log_2(m/2+1))} \\ \mu_{collective} &= \sqrt{2}\sqrt{\frac{1}{m}(ln(1/\epsilon_1) + \Sigma ln(m+1))} \end{aligned} \tag{B10}$$

where Σ is the number of outcomes of POVMs, and m is the total number of signals used for testing.

The step 1 problem becomes

$$\begin{aligned} variable \quad & \Delta\rho \\ variables \quad & \Delta^+, \Delta^-, \delta \\ minimize \quad & Tr((\Delta\rho)^T \nabla f(\rho_i)) \\ Tr(\Gamma_i(\rho_i + \Delta\rho)) &= \gamma_i \\ \Phi_{\mathcal{P}}(\rho_i + \Delta\rho) - \delta &= \vec{F} \\ Tr(\Delta^+) + Tr(\Delta^-) &\leq \mu \\ \Delta^+ &\geq \delta \\ \Delta^- &\geq -\delta \\ (\rho_i + \Delta\rho) &\in \mathbf{H}_+ \end{aligned} \tag{B11}$$

where μ is a bound on the amount of statistical fluctuation acting on all POVMs together, and the map $\Phi_{\mathcal{P}}(\rho) = \sum_j Tr(\rho\tilde{\Gamma}_j)\left|j\right\rangle\left\langle j\right|$, where $\tilde{\Gamma}_j$ are the POVMs whose measurement results will have fluctuations, while Γ_i are the POVMs that have non-fluctuating results (e.g. Alice's source characterization, or normalization operator).

The step 2 problem becomes

$$\begin{aligned} variables \quad & \vec{y}, \vec{z}, a \\ maximize \quad & \vec{\gamma} \cdot \vec{y} + \vec{\gamma} \cdot \vec{y} - a\mu \\ \sum_i y_i \Gamma_i^T + \sum_j z_j \Gamma_j^T &\leq \nabla f(\rho) \\ -a\mathbf{1} \leq \vec{z} &\leq a\mathbf{1} \\ a &\geq 0 \\ \vec{y} &\leq 0 \end{aligned} \tag{B12}$$

The final key rate also needs to be corrected, where (e.g. for collective attack)

$$\begin{aligned} \delta &= 2log_2(5)\sqrt{\frac{log_2(2/\epsilon_2)}{n}} \\ \text{correction} &= \frac{1}{N}(log_2(2/\epsilon_3) - 2log_2(1/\epsilon_4)) \\ R &= (1 - p_{test})(f/ln(2) - \delta) - \text{correction} - (1 - p_{test}) * \text{leak}_{obs}^{EC} \end{aligned} \tag{B13}$$

where N is the total signals sent, p_{test} is the proportion of signals chosen for testing, and f is the solution for either step 1 (for upper bound) or step 2 (for lower bound).

Ref. [2] also proposes two additional new ideas, (1) multiple levels of coarse/fine-graining can be used simultaneously to provide a tighter bound on the statistical fluctuation. (2) there can be an "acceptance criteria" which specifies a threshold over which the protocol is aborted. For simplicity in the code here we consider "unique acceptance" (i.e. always accept frequencies observed). Also, in the current solver we are using a single set of coarse/fine-grained POVMs, but a multiple-graining version can be added in the future.

Appendix C: Documentation: Protocols and Channel Models

In this section we will introduce the example protocols that we provide in this software package, which are based on [1–4, 12]. As mentioned in Sec. III.A, protocol/channel descriptions have the standard format of observables/expectations, kraus operators, and key maps.

There are three example protocol & channel descriptions:

- *Standard BB84*: A prepare-and-measure scheme where Alice encodes her bit in four states from two bases, to be measured by receiver Bob. We include a version using single-photon source with a qubit squashing model for detection, as in Refs. [1, 2], and with a lossy & misaligned channel model, compatible with asymptotic and

finite-size solvers. We also include a version with WCP source and decoy states, as in Ref. [3], compatible with asymptotic solver (decoy-state version).

- *Measurement-device-independent QKD*: A measurement-device-independent scheme where both Alice and Bob encodes bit in signals to send to a third-party Charlie. We include a version with single-photon source as in Ref. [1] compatible with asymptotic and finite-size solvers. We also include a version with WCP source and decoy states, as in Ref. [3], compatible with asymptotic solver (decoy-state version).

- *Discrete-Modulated Continuous Variable QKD (DM-CVQKD)*: A prepare-and-measure CVQKD scheme where Alice encodes in discrete-modulated coherent states, and Bob performs heterodyne measurements. The channel includes loss and excess noise. The description and channel model are based on Ref. [4, 12]. This protocol description is compatible with the asymptotic solver.

Below we will briefly outline the descriptions and the channel models for the protocols used, which are simply recapitulations of the corresponding references on using the numerical framework, including BB84, MDI-QKD [1, 3], and DM-CVQKD [4, 12], which we include for the convenience of the readers. For more details (and derivations) please kindly refer to the original references.

1. BB84

Here we consider the standard BB84 protocol [8], which corresponds to the description files **pmBB84LossyDescription** with **pmBB84LossyChannel**, and with **pmBB84WCPChannel** if decoy-state is used. The protocol description is independent of the encoding degree-of-freedom, although in the lossy channel simulation we assume a polarization-encoding scheme. The BB84 protocol descriptions here are based on Ref. [1]. The actual formulae here are a recapitulation of Ref. [3] Appendix A.1.

In the protocol description file, we need to specify the Kraus operators, key maps, and Alice and Bob's POVMs.

Alice prepares a state in the form of a local qubit entangled with the flying qubit to be sent to Bob:

$$|\psi\rangle_{AA'} = (|0\rangle|H\rangle + |1\rangle|V\rangle + |2\rangle|+\rangle + |3\rangle|-\rangle)/2 \tag{C1}$$

Bob uses a qubit-squashing model [14] (that also contains no-detection event), and his POVMs can be written as:

$$P_1 = 0 \oplus p_Z|H\rangle\langle H|, \quad P_2 = 0 \oplus p_Z|V\rangle\langle V|, \quad P_3 = 0 \oplus p_X|+\rangle\langle +|, \quad P_4 = 0 \oplus p_X|-\rangle\langle -|, \quad P_5 = \mathbb{I} - \sum_{i=1}^{4} P_i \tag{C2}$$

or in terms of matrix entries,

$$P_1^B = p_Z \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad P_2^B = p_Z \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad P_3^B = \frac{1}{2} p_X \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \quad P_4^B = \frac{1}{2} p_X \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{pmatrix}, \quad P_5^B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{C3}$$

In addition to the above observables, we also have some additional constraints that come from Alice's knowledge of what state she sent, in the form of

$$|i\rangle\langle j|_A \otimes \mathbb{I}_{dim_B}, \tag{C4}$$

The Kraus operators are:

		Bob's detectors (passive detection)				Bob's detectors (active detection)			
		H	V	+	-	H	V	+	-
Alice sends	H	$\sqrt{p_Z}\,cos\,\theta$	$\sqrt{p_Z}\,sin\,\theta$	$\sqrt{p_X}\,cos\,\alpha$	$\sqrt{p_X}\,sin\,\alpha$	$cos\,\theta$	$sin\,\theta$	$cos\,\alpha$	$sin\,\alpha$
	V	$-\sqrt{p_Z}\,sin\,\theta$	$\sqrt{p_Z}\,cos\,\theta$	$\sqrt{p_X}\,sin\,\alpha$	$-\sqrt{p_X}\,cos\,\alpha$	$-sin\,\theta$	$cos\,\theta$	$sin\,\alpha$	$-cos\,\alpha$
	+	$\sqrt{p_Z}\,sin\,\alpha$	$\sqrt{p_Z}\,cos\,\alpha$	$\sqrt{p_X}\,cos\,\theta$	$-\sqrt{p_X}\,sin\,\theta$	$sin\,\alpha$	$cos\,\alpha$	$cos\,\theta$	$-sin\,\theta$
	-	$\sqrt{p_Z}\,cos\,\alpha$	$-\sqrt{p_Z}\,sin\,\alpha$	$\sqrt{p_X}\,sin\,\theta$	$\sqrt{p_X}\,cos\,\theta$	$cos\,\alpha$	$-sin\,\alpha$	$sin\,\theta$	$cos\,\theta$

TABLE V. Reproduced from Ref. [3] Table III. The amplitudes of coherent states arriving at Bob's detector, for each given state that Alice sent. Here, we denote $\alpha = \pi/4 - \theta$, and p_Z, p_X as basis choice probability for Bob. For a coherent light source, all terms should be multiplied by $\sqrt{\mu\eta}$ where μ is the intensity and η the channel transmittance. The passive part of the table can also be used for a single-photon source - all terms should be multiplied by $\sqrt{\eta}$, and the square of each term is simply the probability that the photon arrives at that detector.

$$
\begin{aligned}
K_Z &= \left[\begin{pmatrix}1\\0\end{pmatrix}\otimes\begin{pmatrix}1&&\\&0&\\&&0\end{pmatrix}+\begin{pmatrix}0\\1\end{pmatrix}\otimes\begin{pmatrix}0&&\\&1&\\&&0\end{pmatrix}\right]\otimes\sqrt{p_Z}\begin{pmatrix}1&&\\&1&\\&&1\end{pmatrix}\otimes\begin{pmatrix}1\\0\end{pmatrix}\\
K_X &= \left[\begin{pmatrix}1\\0\end{pmatrix}\otimes\begin{pmatrix}0&&\\&0&\\&&1\end{pmatrix}+\begin{pmatrix}0\\1\end{pmatrix}\otimes\begin{pmatrix}0&&\\&0&\\&&0\end{pmatrix}\right]\otimes\sqrt{p_X}\begin{pmatrix}1&&\\&1&\\&&1\end{pmatrix}\otimes\begin{pmatrix}0\\1\end{pmatrix}
\end{aligned}
\tag{C5}
$$

and the key maps are:

$$
\begin{aligned}
Z_1 &= \begin{pmatrix}1&0\\0&0\end{pmatrix}\otimes\mathbb{I}_{dim_A\times dim_B\times 2}\\
Z_2 &= \begin{pmatrix}0&0\\0&1\end{pmatrix}\otimes\mathbb{I}_{dim_A\times dim_B\times 2}
\end{aligned}
\tag{C6}
$$

Secondly, we can consider the channel model, which generates the γ constraints corresponding to each POVM. Note that, the channel model is independent of the above protocol description, and different channel models can be swapped as long as they generate the corresponding statistics for each POVM.

Here we include the lossy and misaligned channel model as described in Ref. [3], and provide a recapitulation for reference purposes.

We can either assume a single-photon source (for file **pmBB84LossyChannel**) or a phase-randomized weak coherent pulse source (for file **pmBB84WCPChannel**, which needs to be combined with decoy-state analysis as we will later discuss).

The misaligned and lossy channel can be described by Table V, which represents the arriving probability for a single-photon, or the arriving amplitudes of coherent lights at each detector,

For a single-photon source, these arriving probabilities (if there is no dark count) can be used to directly generate the expectation values of the POVMs.

For a WCP source, the amplitudes can be used to calculate the detector click probability:

$$
p_{j|i}^{\text{click}} = 1 - (1 - p_d) \times e^{-|\alpha_{j|i}|^2}. \tag{C7}
$$

where p_d is the dark count probability, and $\alpha_{j|i}$ is the amplitude in Table V conditional to Alice's choice of state. Once the detection probability is obtained, we can combine the probabilities for the detectors (for a passive detection setup):

$$
p_{b_1b_2b_3b_4|i} = \Pi_{j=1,2,3,4}\left\{\overline{b_j} + p_{j|i}^{\text{click}}\,(-1)^{\overline{b_j}}\right\} \tag{C8}
$$

where $b_1b_2b_3b_4$ is the detection pattern (each bit is 0 if the detector doesn't click, and 1 if it clicks), for detectors $H, V, +, -$. $\overline{b_j}$ is the bit flip of b_j. The above equation is simply describing that the joint detection pattern probability is the product of four terms, each term is either $p^{\text{click}}_{j|i}$ if $b_j = 1$, or $1 - p^{\text{click}}_{j|i}$ if $b_j = 0$.

Alternatively, for an active detection setup,

$$\begin{aligned}
p^{\text{click}}_{\text{basis},j|i} &= b_{\text{basis},j}\,[1-(1-p_d)\times e^{-|\alpha_{j|i}|^2}],\\
p^{Z}_{b_1b_2b_3b_4|i} &= \Pi_{j=1,2,3,4}\,\{\overline{b_j} + p^{\text{click}}_{Z,j|i}(-1)^{\overline{b_j}}\},\\
p^{X}_{b_1b_2b_3b_4|i} &= \Pi_{j=1,2,3,4}\,\{\overline{b_j} + p^{\text{click}}_{X,j|i}(-1)^{\overline{b_j}}\},\\
p_{b_1b_2b_3b_4|i} &= p_Z\, p^{Z}_{b_1b_2b_3b_4|i} + p_X\, p^{X}_{b_1b_2b_3b_4|i},
\end{aligned} \tag{C9}$$

where the difference is that either only Z detectors or only X detectors are allowed to activate at a time, depending on the basis choice . Again we define a logical bit $b_{\text{basis},j}$, which is 1 if the chosen basis agrees with the detector j (H and V for Z basis, or $+$ and $-$ for X basis), or 0 if it disagrees. The final pattern probability is the weighted sum of the two conditional probabilities for Z and X bases.

After the pattern probability (a total of 16 patterns for 4 detectors) is calculated, we need to perform coarse-graining to map it to 5 POVMs, because we used a qubit-squashing model:

$$\begin{aligned}
M_H &= [0,0,0,0,0,0,0,0,1,0,0,0,0.5,0,0,0]\\
M_V &= [0,0,0,0,1,0,0,0,0,0,0,0,0.5,0,0,0]\\
M_+ &= [0,0,1,0.5,0,0,0,0,0,0,0,0,0,0,0,0]\\
M_- &= [0,1,0,0.5,0,0,0,0,0,0,0,0,0,0,0,0]\\
M_\varnothing &= [1,0,0,0,0,1,1,1,0,1,1,1,0,1,1,1]\\
M &= [M_H^T, M_V^T, M_+^T, M_-^T, M_\varnothing^T]
\end{aligned} \tag{C10}$$

Note that the above pattern probabilities are the raw statistics coming from WCP sources, while to calculate the secure key we need to obtain the single-photon contribution among these data. Therefore, we need to perform decoy-state analysis [15–17], either before or after applying the above coarse-graining.

For conciseness, we will not go into details on performing the decoy-state analysis, which is described in many works, and Ref. [3] also includes a review of the procedure. The key idea is that, WCP sources send a mixture of photon number states following a Poissonian photon number distribution

$$p_{\mu_i}(n) = \frac{{\mu_i}^n}{n!}\, e^{-\mu_i}. \tag{C11}$$

and the expectation values we observe from experiments is a sum of the contribution of all photon number states:

$$\gamma_{\mu_i} = \sum p_{\mu_i}(n)\,\gamma_n. \tag{C12}$$

This is actually a linear equation, and if we use a set of different intensities $\{\mu_i\}$, we can obtain a set of linear equations with $\{\gamma_n\}$ as variables, and the experimental observables $\{\gamma_{\mu_i}\}$ as constraints. This is a linear programming problem [18] and can be solved by e.g. CVX, where we would like to find the upper and lower bounds on the single-photon contribution γ_1, denoted as γ_1^L, γ_1^U. The linear program modeling is embedded in the channel file **pmBB84WCPChannel**.

In an actual program, we cannot define an infinite photon number space, and the common practice is to implement a cutoff N on the photon number, where all statistics above this point are upper bounded by 1 and lower bounded by 0:

$$\begin{aligned}
\gamma_{\mu_i} &\le \sum_{n\le N} p_\mu(n)\,\gamma_n + (1 - \sum_{n\le N} p_\mu(n)),\\
\gamma_{\mu_i} &\ge \sum_{n\le N} p_\mu(n)\,\gamma_n.
\end{aligned} \tag{C13}$$

The above procedure is described for a given observable γ. In practice, we need to perform one decoy-state analysis on each one of the observables $\{\gamma_k\}$ to obtain upper and lower bounds on it, $\gamma^L_{1,k}, \gamma^U_{1,k}$, for 4×5 times if we apply decoy-state analysis after coarse-graining, or 4×16 times if before. The decoy-state asymptotic solver would formulate the constraints for the density matrix ρ as:

$$\gamma^L_{1,k} \leq Tr(\Gamma_k \, \rho) \leq \gamma^U_{1,k}, \; \forall k. \tag{C14}$$

2. Discrete-Modulated Continuous Variable QKD

This subsection is a recapitulation of Ref. [4, 12].

In the previous subsections, the protocols we have introduced are all based on single photon detectors and single photon sources (or weak coherent pulse sources whose single photon component can be estimated). In Continuous Variable (CV) QKD, the sources are coherent states modulated in phase space, and homodyne/heterodyne detections are performed (that observes a certain component or region in phase space). Since its based on coherent states instead of qubits, it is named continuous variable QKD (in contrast to the traditional discrete varialble QKD). The key advantage of CVQKD is it does not require single photon detectors or single photon sources, and is compatible with the existing infrastructure designed for classical communications.

In discrete-modulated CVQKD (in contrast to e.g. Gaussian-modulated CVQKD, where user encodes based on random variables sampled from Gaussian distributions) Alice encodes in four states $\{|\alpha\rangle, |-\alpha\rangle, |i\alpha\rangle, |-i\alpha\rangle\}$:

$$|\psi\rangle_{AA'} = (|0\rangle |\alpha\rangle + |1\rangle |-\alpha\rangle + |2\rangle |i\alpha\rangle + |3\rangle |-i\alpha\rangle)/2 \tag{C15}$$

Bob can choose to use either homodyne detection or heterodyne detection. In the example included in this software package, we consider the heterodyne detection case. To generate key, Bob divides the phase space into four regions (separated by intersection areas of Δ_p between each two regions where results are inconclusive). Such an operation of distinguishing the four regions is described by

$$\begin{aligned}
R_0 &= \frac{1}{\pi} \int_{\Delta_a}^{\infty} \int_{-(\pi/4)+\Delta_p}^{(\pi/4)-\Delta_p} \gamma \, |\gamma e^{i\theta}\rangle \langle \gamma e^{i\theta}| \, d\theta d\gamma \\
R_1 &= \frac{1}{\pi} \int_{\Delta_a}^{\infty} \int_{(\pi/4)+\Delta_p}^{(3\pi/4)-\Delta_p} \gamma \, |\gamma e^{i\theta}\rangle \langle \gamma e^{i\theta}| \, d\theta d\gamma \\
R_2 &= \frac{1}{\pi} \int_{\Delta_a}^{\infty} \int_{(3\pi/4)+\Delta_p}^{(5\pi/4)-\Delta_p} \gamma \, |\gamma e^{i\theta}\rangle \langle \gamma e^{i\theta}| \, d\theta d\gamma \\
R_3 &= \frac{1}{\pi} \int_{\Delta_a}^{\infty} \int_{(5\pi/4)+\Delta_p}^{(7\pi/4)-\Delta_p} \gamma \, |\gamma e^{i\theta}\rangle \langle \gamma e^{i\theta}| \, d\theta d\gamma
\end{aligned} \tag{C16}$$

and the Kraus operators can be written as

$$K = \sum_{z=0}^{3} |z\rangle_R \otimes \mathbb{I}_A \otimes (\sqrt{R_z})_B \tag{C17}$$

where R is the register storing bit result, and A,B are respectively Alice's and Bob's systems. The postprocessing map is calculated by $\mathcal{G}(\rho) = K \rho K^\dagger$.

The keymap simply retrieves the register R, and can be written as

$$Z_j = |j\rangle \langle j|_R \otimes \mathbb{I}_A \otimes \mathbb{I}_B \tag{C18}$$

For the observables, Bob's measurement can be described by the Husimi Q function $Q(\gamma) = (1/\pi) \langle \gamma | \, \rho \, | \gamma \rangle = Tr(\rho E_\gamma)$, where E_γ is the POVM operator, and any operator described by creation and annihilation operators can be expressed as (quoting Ref. [4] Eq. 19)

$$Tr(\rho\hat{f}(\hat{a},\hat{a}^\dagger)) = \int d^2\gamma Q(\gamma) f^{(A)}(\gamma) \tag{C19}$$

where $\hat{f}(\hat{a},\hat{a}^\dagger)$ can be among the set of

$$\begin{aligned}
\hat{q} &= \frac{1}{\sqrt{2}}(\hat{a}+\hat{a}^\dagger) \\
\hat{p} &= \frac{i}{\sqrt{2}}(\hat{a}-\hat{a}^\dagger) \\
\hat{n} &= \hat{a}^\dagger\hat{a}^\dagger \\
\hat{d} &= \hat{a}^2 + (\hat{a}^\dagger)^2
\end{aligned} \tag{C20}$$

and $f^{(A)}(\gamma)$ is where $\hat{a},\hat{a}^\dagger$ is replaced by γ,γ^*.

Similar to the aforementioned protocols, Alice should also force a constraint on the prepared state $\rho_A = Tr_B(\rho)$ (as she can characterize her own trusted source), such that

$$\rho_A = \sum \sqrt{p_x p_{x'}} \left\langle \phi_{x'} | \phi_x \right\rangle_{A'} |x\rangle \langle x'|_A \tag{C21}$$

where $|\phi_x\rangle$ is the encoded coherent state and $|x\rangle$ is Alice's local qubit. p_x is the probability of choosing a given state (here it is 1/4 for each of the four states).

Now, for the channel model simulation, conditionally for each given state x prepared (corresponding to the $|x\rangle\langle x|$ measurement on Alice's local qubit), the expected values for Bob's measurements can be shown as

$$\begin{aligned}
Tr(\rho\hat{q})_x &= \sqrt{2\eta} Re(\alpha_x) \\
Tr(\rho\hat{p})_x &= \sqrt{2\eta} Im(\alpha_x) \\
Tr(\rho\hat{n})_x &= \eta|\alpha_x|^2 + \frac{\eta\xi}{2} \\
Tr(\rho\hat{d})_x &= \eta[\alpha_x^2 + (\alpha_x^*)^2]
\end{aligned} \tag{C22}$$

where the channel is modeled by the loss η and excess noise ξ, and the source described by its amplitude α.

The observed probability distribution (used for error correction) is

$$P(z=j|x=k) = \int_{\Delta_a}^{\infty} \int_{((2j-1)/4)+\Delta_p}^{((2j+1)\pi/4)-\Delta_p} \frac{exp(-\frac{|\gamma e^{i\theta} - \sqrt{\eta}\alpha_k|^2}{1+\eta\xi/2})}{\pi(1+\eta\xi/2)} \gamma d\theta d\gamma \tag{C23}$$

3. Measurement-Device-Independent BB84

This subsection describes the measurement-device-independent QKD protocol, based on [11], whose numerical form is described in [9]. The protocol description corresponds to the file **mdiBB84Description**. The actual formulae are a recapitulation of Appendix A.2 in Ref. [18].

Up so far we have discussed two-party protocols where Alice sends signals to Bob (and, in the security proof, keeps half of the entangled system to herself). As described in Ref. [9], the framework is in principle applicable to a three-party system too. measurement-device-independent (MDI) QKD is one of such protocols, where Alice and Bob both prepare and send signals, to an untrusted third-party Charlie, who performs measurements and publicly announces the results. Importantly, Charles only knows the parity between Alice's and Bob's signals, and not the actual bits, allowing Charles to be untrusted, thus making the MDI-QKD protocol immune to all detector side-channels.

An example setup can be found in Fig. 4, where Alice and Bob both prepare a local qubit entangled with the flying qubits, and Charles performs a Bell State Measurement (BSM) on the flying qubits. Note that in MDI-QKD,

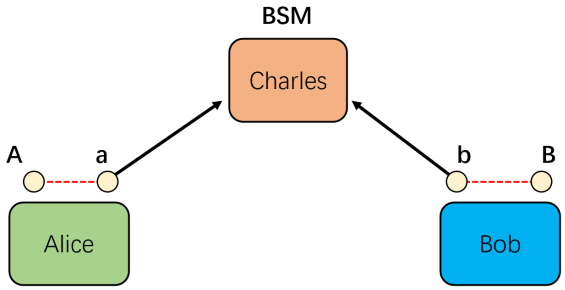

FIG. 4. Illustration of the setup for MDI-QKD. Here Alice and Bob both prepare local system A,B, and send entangled qubits a,b to a third-party relay Charles, who performs a bell state measurement (BSM) and announces the results publicly.

Charles can only distinguish between two Bell states $|\Phi^{\pm}\rangle = (|HV\rangle \pm |VH\rangle)/\sqrt{2}$, and the other cases are considered inconclusive. Charles' POVMs are:

$$P_1^C = |\Phi^+\rangle_{ab}\langle\Phi^+|_{ab}, \quad P_2^C = |\Phi^-\rangle_{ab}\langle\Phi^-|_{ab} \quad P_3^C = 1 - P_1^C - P_2^C \tag{C24}$$

The overall POVM is constructed in the form of:

$$\Gamma_{ijk} = P_i^A \otimes P_j^B \otimes P_k^C \tag{C25}$$

Like for BB84, here we should also include the deterministic constraints representing Alice and Bob's characterization of their sources:

$$|i\rangle\langle j|_A \otimes |k\rangle\langle l|_B \otimes \mathbb{I}_{dim_C} \tag{C26}$$

The Kraus operators are now:

$$\begin{aligned} K_Z &= \left[\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1 & & \\ & 0 & \\ & & 0\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0 & & \\ & 1 & \\ & & 0\end{pmatrix}\right] \otimes \begin{pmatrix}1 & & \\ & 1 & \\ & & 0\end{pmatrix} \otimes \begin{pmatrix}1 & & \\ & 1 & \\ & & 0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} \\ K_X &= \left[\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0 & & \\ & 0 & \\ & & 1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0 & & \\ & 0 & \\ & & 1\end{pmatrix}\right] \otimes \begin{pmatrix}0 & & \\ & 0 & \\ & & 1\end{pmatrix} \otimes \begin{pmatrix}1 & & \\ & 1 & \\ & & 0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \end{aligned} \tag{C27}$$

and the key maps are:

$$\begin{aligned} Z_1 &= \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} \otimes \mathbb{I}_{dim_A \times dim_B \times dim_C \times 2} \\ Z_2 &= \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} \otimes \mathbb{I}_{dim_A \times dim_B \times dim_C \times 2} \end{aligned} \tag{C28}$$

note that here the dimensions are $\dim_A = \dim_B = 4$ and $\dim_C = 3$. Because Charles announces all results publicly, one can consider his system as classical (which makes ρ block diagonal with respect to the three subspaces conditional to the three outcomes Charles announces).

Secondly, we consider the channel model. Here we first discuss a simple model of a dephasing channel and single-photon sources **MDIBB84DepolarizingChannel**, and then we will describe a realistic model based on a lossy and

misaligned channel, for phase-randomized WCP sources, in **MDIBB84LossyChannel** and **MDIBB84WCPChannel**.

For the simple case with single-photons, Alice and Bob prepare the state:

$$\begin{aligned}
|\psi\rangle_{Aa} &= (\sqrt{p_Z}\,|0\rangle\,|H\rangle + \sqrt{p_Z}\,|1\rangle\,|V\rangle + \sqrt{p_X}\,|2\rangle\,|+\rangle + \sqrt{p_X}\,|3\rangle\,|-\rangle)/\sqrt{2} \\
|\psi\rangle_{Bb} &= (\sqrt{p_Z}\,|0\rangle\,|H\rangle + \sqrt{p_Z}\,|1\rangle\,|V\rangle + \sqrt{p_X}\,|2\rangle\,|+\rangle + \sqrt{p_X}\,|3\rangle\,|-\rangle)/\sqrt{2} \\
\rho_{AaBb} &= |\psi\rangle_{Aa}\,|\psi\rangle_{Ba}\,\langle\psi|_{Aa}\,\langle\psi|_{Ba}
\end{aligned} \tag{C29}$$

We consider a depolarizing channel:

$$\begin{aligned}
Z_a^I &= [\sqrt{1-(3/4)dp}]I_A \otimes I_B \otimes I_a \otimes I_b \\
Z_a^X &= (\sqrt{dp}/2)I_A \otimes I_B \otimes \sigma_{X,a} \otimes I_b \\
Z_a^Y &= (\sqrt{dp}/2)I_A \otimes I_B \otimes \sigma_{Y,a} \otimes I_b \\
Z_a^Z &= (\sqrt{dp}/2)I_A \otimes I_B \otimes \sigma_{Z,a} \otimes I_b
\end{aligned} \tag{C30}$$

$$\begin{aligned}
Z_b^I &= [\sqrt{1-(3/4)dp}]I_A \otimes I_B \otimes I_a \otimes I_b \\
Z_b^X &= (\sqrt{dp}/2)I_A \otimes I_B \otimes I_a \otimes \sigma_{X,b} \\
Z_b^Y &= (\sqrt{dp}/2)I_A \otimes I_B \otimes I_a \otimes \sigma_{Y,b} \\
Z_b^Z &= (\sqrt{dp}/2)I_A \otimes I_B \otimes I_a \otimes \sigma_{Z,b}
\end{aligned}$$

where $\sigma_X, \sigma_Y, \sigma_Z$ are the Pauli matrices. With the transformations, we can write the density matrix after the channel as

$$\rho_{Aa'Bb'} = \sum_{ij} Z^{ij}\rho_{AaBb}Z^{ij\prime} \tag{C31}$$

where

$$Z^{ij} = Z_a^i Z_b^j \tag{C32}$$

At this point we can simulate the statistics using

$$\gamma_i = Tr(\rho_{Aa'Bb'}\Gamma_i') \tag{C33}$$

and proceed to calculate the key rate.

For a realistic channel with misalignment and loss, we do a recapitulation of results from Ref. [3]. The details of the model and simulation results can be found in this reference. Also, the channel model for MDI-QKD has been discussed in Ref. [19, 20], and the WCP interference model is discussed in e.g. Ref. [21].

Similar to the case of BB84, we can simulate the amplitudes of the coherent states arriving at Charles:

$$\begin{aligned}
\alpha_{3H}^{\phi} &= \sqrt{\mu_A\,\eta_A\cos\theta_A/2} + i\sqrt{\mu_B\,\eta_B\cos\theta_B/2}\,e^{i\phi}, \\
\alpha_{4H}^{\phi} &= i\sqrt{\mu_A\,\eta_A\cos\theta_A/2} + \sqrt{\mu_B\,\eta_B\cos\theta_B/2}\,e^{i\phi}, \\
\alpha_{3V}^{\phi} &= \sqrt{\mu_A\,\eta_A\sin\theta_A/2} + i\sqrt{\mu_B\,\eta_B\sin\theta_B/2}\,e^{i\phi}, \\
\alpha_{4V}^{\phi} &= i\sqrt{\mu_A\,\eta_A\sin\theta_A/2} + \sqrt{\mu_B\,\eta_B\sin\theta_B/2}\,e^{i\phi},
\end{aligned} \tag{C34}$$

where the four detectors are behind the two output arms $3, 4$ of Charles' beam splitter, and use polarizing beam splitters (PBS) to measure the polarization of the signals. Here we assume Alice and Bob prepare states in polarization-encoding θ_A, θ_B, which could have also gone through some misalignment before arriving at Charles (which means the

misalignment angle would be added into θ_A, θ_B too). The channel transmittances are η_A, η_B, and source intensities are μ_A, μ_B. Note that, importantly, Charles' performs an interference of Alice's and Bob's signals, and the outcome amplitudes depends on the relative phase ϕ between Alice and Bob's signals, which is why we denote all amplitudes as conditional to ϕ.

Once we obtain the amplitudes, just like in BB84, we can calculate the click probabilities:

$$p_{k|ij}^{\text{click},\phi} = 1 - (1 - p_d) \times e^{-|\alpha_{k|ij}^{\phi}|^2} \tag{C35}$$

where k is the index of a given detector. Again, we can calculate the detector pattern probability:

$$p_{b_1 b_2 b_3 b_4|ij}^{\phi} = \prod_{k=1,2,3,4} \{\overline{b_k} + p_{k|ij}^{\text{click}} \, (-1)^{\overline{b_k}}\} \tag{C36}$$

for each pattern $b_1 b_2 b_3 b_4$.

Importantly, here Alice and Bob both use phase-randomized sources, and the relative phase between them is evenly distributed between $[0, 2\pi)$. Therefore, we must integrate all statistics over $\phi \in [0, 2\pi)$:

$$p_{b_1 b_2 b_3 b_4|ij} = 1/2\pi \int_0^{2\pi} p_{b_1 b_2 b_3 b_4|ij}^{\phi} \, d\phi. \tag{C37}$$

The statistics after integrating (16 patterns, for 16 combinations of states that Alice and Bob sent) constitute the raw statistics. Again, we can apply a coarse-graining map:

$$\begin{aligned} M_{\Psi^-} &= [0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0] \\ M_{\Psi^+} &= [0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0] \\ M_{\varnothing} &= 1_{1\times 16} - M_{\Psi^+} - M_{\Psi^-} \\ M &= [M_{\Psi^-}^T, M_{\Psi^+}^T, M_{\varnothing}^T] \end{aligned} \tag{C38}$$

to map it to correspond to Charles' three POVMs. Note that, since we are not using a squashing model for MDI-QKD, we can in principle use all raw statistics as constraints for the solver, and this coarse-graining is optional. Nonetheless, we perform this coarse-graining here to reduce the number of constraints and thus the computational complexity.

Again, just like in BB84, we can apply decoy-state analysis (either before or after the coarse-graining) to obtain the single-photon contribution among them.

The photon number distribution now becomes a product of Alice's and Bob's distributions, and the observables is a mixture of contributions of photon number states (n_A, n_B):

$$\gamma_{\mu_{Ai},\mu_{Bj}} = \sum_{n_A} \sum_{n_B} p_{\mu_{Ai}}(n_A) \, p_{\mu_{Bj}}(n_B) \, \gamma_{n_A,n_B} \tag{C39}$$

Again, to make the problem finite, we implement photon number cutoff S on the terms, and upper (lower) bound all terms exceeding the cutoff by 1 (0):

$$\begin{aligned} \gamma_{\mu_i} \leq & \sum_{n_A \leq N} \sum_{n_B \leq N} p_{\mu_{Ai}}(n_A) \, p_{\mu_{Bj}}(n_B) \, \gamma_{n_A,n_B} \\ & + (1 - \sum_{n_A \leq N} \sum_{n_B \leq N} p_{\mu_{Ai}}(n_A) \, p_{\mu_{Bj}}(n_B)), \\ \gamma_{\mu_i} \geq & \sum_{n_A \leq N} \sum_{n_B \leq N} p_{\mu_{Ai}}(n_A) \, p_{\mu_{Bj}}(n_B) \, \gamma_{n_A,n_B}. \end{aligned} \tag{C40}$$

At this point, we can feed the problem to a linear solver (modeling embedded in **MDIBB84WCPChannel**) to solver for the bounds on single-photon contributions, and use them as constraints for the decoy-state asymptotic solver.

Appendix D: Documentation: Error Correction Leakage

In this section we briefly discuss the error correction leakage calculation we use.

In the simpler case, Alice and Bob can consider binary "correct/error" statistics (which can be for multiple sets of data, e.g. X and Z bases), and obtain the error rate E for each set. The leakage is

$$\sum p_{pass,i}\, f_{EC}\, h_2(E_i) \tag{D1}$$

where h_2 is the binary entropy function, $p_{pass,i}$ is the probability of obtaining data in the specific set (including sifting and channel loss), and f_{EC} is the error correction efficiency (a value larger than or equal to 1).

For a tighter calculation, Alice and Bob can also use the full statistics (a probability distribution of all Alice's measurements X, i.e. prepared states, versus Bob's results Y), which we can denote as P_{XY}. If we only look at one system, e.g. X (Y), and sum over all Y (X) entries, we can get the distribution P_X (P_Y). We can then calculate

$$\begin{aligned} H(XY) &= -\sum_{P_i>0} P_{XY,i}\, log P_{XY,i} \\ H(X) &= -\sum_{P_{X,i}>0} P_{X,i}\, log P_{X,i} \\ H(Y) &= -\sum_{P_{Y,i}>0} P_{Y,i}\, log P_{Y,i} \end{aligned} \tag{D2}$$

from which we can obtain

$$\begin{aligned} H(Y|X) &= H(XY) - H(X) \\ H(X|Y) &= H(XY) - H(Y) \\ I(XY) &= H(Y) - H(Y|X) \end{aligned} \tag{D3}$$

The leakage (before applying sifting factor and gain, p_{pass}) is

$$\text{leak}_{obs}^{EC} = f_{EC} H(X|Y) \tag{D4}$$

Note that the error-correction model is conventionally slightly different for CVQKD:

$$\begin{aligned} \text{leak}_{obs}^{EC} &= (1-\beta) I(XY) + \beta H(X|Y) \quad (reverse\ \ reconciliation) \\ \text{leak}_{obs}^{EC} &= (1-\beta) I(XY) + \beta H(Y|X) \quad (direct\ \ reconciliation) \end{aligned} \tag{D5}$$

where error correction efficiency is defined as $\beta \leq 1$. (In this software package, however, when choosing a CVQKD protocol, to maintain a consistent interface, f will automatically be interpreted as β, and the user can simply fill in a value of $f \leq 1$.)

Appendix E: Documentation: Parameter Optimization Algorithms

In this section we introduce the parameter optimization algorithms used in the software. Let us denote all the optimizable parameters as a vector $\vec{p}$, and the key rate $f(\vec{p})$ is a single-valued function of $\vec{p}$. The goal is to find an optimal $\vec{p}$ that maximizes the key rate function.The optimizer module is completely decoupled from the protocol and solvers, and treats the key rate function as a black box.

- *Brute-force Search*: In this algorithm we can simply search over the entire parameter space for all possible components of $\vec{p}$, subject to a finite resolution. This is effective for smaller numbers of parameters and coarse resolutions, but the search time exponentially grows with the number of parameters.

- *Coordinate Descent*: Algorithm used in Ref. [5, 6]. Each component (i.e. coordinate) of $\vec{p}$ is updated one at a time while keeping other components fixed.

$$p_k^{i+1} = argmax_{p_k} f(p_1^{i+1}, p_2^{i+1}, ..., p_{k-1}^{i+1}, p_k, p_{k+1}^{i}, ..., p_N^{i}) \tag{E1}$$

  where the superscript is for the iteration, and the underscript is for the component index. The linear search is performed with MATLAB's built-in **fminbnd** function. When all components are updated, the algorithm starts over the next iteration from p_1. The iteration continues until maximum iterations are reached or two iterations yield sufficiently close function values. This is a type of local search similar to gradient descent, and its advantage is the possibility of parallelization (for the linear search) and its being less likely to get trapped locally by small numerical noise.

- *Gradient Descent*: A standard gradient descent method. A small sample step in each component direction is taken to evaluate the partial derivaties, which are then combined to form $\nabla f(\vec{p})$. The current position is updated by

$$\vec{p}^{i+1} = \vec{p}^{i} + \alpha \nabla f(\vec{p^i}) \tag{E2}$$

  where α is the learning rate. The algorithm stops when maximum iteration number is reached or if two iterations return close enough values.

- *Adam Optimizer [7]*: An adaptive learning rate local search algorithm commonly used in machine learning fields (but can be used for parameter optimization of a convex function, too). The optimizer is based on momentum-based learning rate, and keeps track of two terms m^i and v^i, the momentum and variance (second order momentum). As described in Ref. [7], the updating of parameters follows:

$$\begin{aligned} m^{i+1} &= \beta_1 m^i + (1-\beta_1)\nabla f(\vec{p^i}) \\ v^{i+1} &= \beta_2 v^i + (1-\beta_2)\nabla^2 f(\vec{p^i}) \\ \hat{m}^{i+1} &= m^{i+1}/(1-\beta_1^{i+1}) \\ \hat{v}^{i+1} &= v^{i+1}/(1-\beta_2^{i+1}) \\ \vec{p}^{i+1} &= \vec{p}^{i} - \alpha \hat{m}^{i+1}/(\sqrt{\hat{v}^{i+1}} + \epsilon) \end{aligned} \tag{E3}$$

  where β_1, β_2 are learning-rate related (damping) factors, commonly taking values close to 1. ϵ is set to be a small constant (e.g. 10^{-8}) to avoid division by zero. The original optimizer is written for descent direction, but simply setting it to optimize $(-R)$ allows one to find $\vec{p}$ that maximizes key rate. Adam optimizer is considered a versatile local search algorithm that works well against difficult landscapes (i.e. function shapes) and converges faster than gradient descent.

[1] A Winick, N Lütkenhaus, PJ Coles. Quantum 2 (2018): 77.
[2] I George, J Lin, N Lütkenhaus. Physical Review Research 3 (2021): 013274.
[3] W Wang, N Lütkenhaus. arXiv:2108.10844 (2021).
[4] J Lin, T Upadhyaya, and N Lütkenhaus. Physical Review X 9 (2019): 041064.
[5] F Xu, H Xu, HK Lo. Physical Review A 89 (2014): 052333.
[6] W Wang, F Xu, and HK Lo. Physical Review X 9 (2019): 041012.
[7] DP Kingma, J Ba. arXiv:1412.6980 (2014).
[8] CH Bennett, G Brassard. Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, 175-179 (1984).
[9] PJ Coles, EM Metodiev, N Lütkenhaus. Nature Communications 7 (2016): 1-9.
[10] D Bruß. Physical Review Letters 81 (1998): 3018.
[11] HK Lo, M Curty, B Qi. Phys. Review Letters 108 (2012): 130503.
[12] T Upadhyaya, T Himbeeck, J Lin, N Lütkenhaus, PRX Quantum 2 (2021): 020325.
[13] R Renner. International Journal of Quantum Information 6 (2008): 1-127.
[14] O Gittsovich, NJ Beaudry, V Narasimhachar, RR Alvarez, T Moroder, N Lütkenhaus. Physical Review A 89 (2014): 012325.
[15] WY Hwang, Physical Review Letters 91 (2003): 057901.

[16] HK Lo, X Ma, and K Chen. Physical review letters 94 (2005): 230504.
[17] XB Wang, Physical Review A 72 (2005): 012322.
[18] P Rice, and J Harrington. arXiv preprint arXiv: 0901.0013 (2008).
[19] X Ma, CH Fung, and M Razavi. Physical Review A 86 (2012): 052305.
[20] F Xu, M Curty, B Qi, HK Lo, New Journal of Physics 15 (2013): 113007.
[21] M Curty, T Moroder, X Ma, N Lütkenhaus, Optics letters 34 (2009): 3238-3240.