Code for: The Influence of Experimental Imperfections on Photonic GHZ State Generation

Fabian Wiesner

Jun 07, 2024

CONTENTS:

 This project uses and requires the Boost Library. In particular, the header files `boost/container/flat_map.hpp` and `boost/algorithm/string.hpp` were imported.

CHAPTER

ONE

INDICES AND TABLES

- genindex

1.1 Table of Contents

1.1.1 Documentation

KeyAux

Some helper functions for Key.hpp.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Copyright
None, this file only contains trivial common knowledge code.

Functions

int **facut**(int i)

Recrusive definition of the faculty. Up to 12, this is just a look-up.

Parameters
i – Input to compute the faculty of.

Returns
int The fuculty of i, i.e. i!.

template<class **R**, class **I**>
R **binomialCoeff**(*I* n, *I* k)

Returns the binomial coefficient n over k. Shouldn't be used for large numbers and is only used for n<=12.

Template Parameters

- **R** – Real-type for the output.

- **I** – Integer number type used for input.

Parameters

- **n** – Upper number of the binomial coefficient, 0<=n<=12.
- **k** – Lower number of the binomial coefficient, 0<=k<=n.

Returns
R n over k.

Variables

const int **FACUT12**[13] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600}

Pre-computed faculties up to 12! as this is the maximal number of photons in the circuit.

namespace **numbers**

Short handle for pi.

Variables

const double **pi** = 3.14159265358979323846

Key

Defines the class *Key* used for the *State* sturcture.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Copyright
Copyright (c) 2024, provided under CC BY-NC 4.0. license

template<class **Int**>

class **Key** : public boost::container::flat_map<std::pair<*Int*, *Int*>, *Int*>

#include <*Key.hpp*> *Key* class, that implements the functions needed for *State*. Inherits from boost::container::flat_map<std::pair<Int, Int>, Int>.

Template Parameters
Int – Class used for integer numbers.

Public Types

using **Par** = boost::container::flat_map<std::pair<*Int*, *Int*>, *Int*>

Short handle for the parent type that provide the actual data structure.

using **basetype** = *Int*

Handle for the intereger number type to provide it for *State*.

template<class **Val**>

using **SD** = boost::container::flat_map<*Key*<*Int*>, *Val*>

Handle for State-like (actually State-parent) data structure.

Template Parameters
Val – Value-type, cf. *State*.

Public Functions

inline **Key**()

Construct a new *Key* object.

inline **Key**(*Int* a, *Int* b, *Int* c)

Construct a new *Key* object from an entry.

Parameters

- **a** – spatial & polarization mode
- **b** – distinguishability mode
- **c** – occupation number

inline void **add**(const *Par* &p)

Adds a boost::container::flat_map<std::pair<Int, Int>, Int>, if entry exist, occupation numbers are added.

Parameters
p – boost::container::flat_map<std::pair<Int, Int>, Int> to add.

inline **operator** *Par*() const

Returns this as boost::container::flat_map<std::pair<Int, Int>, Int>.

Returns
Par, i.e. boost::container::flat_map<std::pair<Int, Int>, Int>

inline void **clean**()

Removes entries with occupation number 0.

template<class **Val**>
inline *Val* **factor**(*Int* a, *Int* b) const

Normalization factor for second-quantization for Spatial&Polarization modes a and b.

Template Parameters
Val – Value-type, cf. *State*

Parameters

- **a** – Spatial&Polarization mode
- **b** – Spatial&Polarization mode

Returns
Val The normalization factor

template<class **Val**>
inline *Val* **factor**() const

Normalization factor for second-quantization for all modes.

Template Parameters
Val – Value-type, cf. *State*

Returns
Val The normalization factor

inline void **addEnd**(const *Int* &a, const *Int* &b, const *Int* &c)

Adds entry and the end.

Parameters

- **a** – Spatial&Polarization mode
- **b** – Distinguishability mode
- **c** – Occupation number

template<class **Val**, class **Real**>
inline *SD*<*Val*> **apply**(const std::vector<*Val*> &U, const std::vector<*Int*> &modes, const *Real* &tol) const

Applies a Unitary in second quantization.

Template Parameters

- **Val** – Amplitude type that is used in the *State*
- **Real** – Real number type that should be used, e.g. float

Parameters

- **U** – Unitary in line format
- **modes** – Affected modes, max 2.
- **tol** – Tolerance for the aplitude, amplitude with absolute value lower than tol are discarded

Returns
SD<Val> *State* similar data structure one gets, if one applies U on the implicit state (*Key* : 1.0).

template<class **Val**>
inline *Val* **apply**(const *Val* &U, const *Int* &mode) const

Applies a Unitary in second quantization for a single mode.

Template Parameters
Val – Amplitude type that is used in the *State*

Parameters

- **U** – Unitary in line format
- **mode** – affected mode

Returns
Val The factor for the amplitude

inline void **swap**(*Int* a, *Int* b)

Swaps Spatial&Polarization modes a and b.

Parameters

- **a** – Spatial&Polarization mode
- **b** – Spatial&Polarization mode

inline bool **overlapping**(const boost::container::flat_map<*Int*, *Int*> &ref) const

Checks if the *Key* containes data of a boost::container::flat_map<Int, Int>.

Parameters
ref – The data to check if contained

Returns
Bool, if contained or not

template<class **Val**>
inline *SD*<*Val*> **loss**(const std::vector<*Int*> &modes, const *Int* &lossMode, *Int* &maxLM) const

Applies loss with uniform probability per photon on the modes.

Template Parameters
Val – Amplitude type that is used in the *State*

Parameters

- **modes** – Spatial&Polarization Modes affected by loss
- **lossMode** – Spatial&Polarization Mode the photon goes to
- **maxLM** – Saves the highest mode used for loss.

Returns
SD<Val>, i.e. boost::container::flat_map<Key<Int>, Val>, that is obtained by loss on (*Key*:1.0)

template<class **Val**>
inline void **collapse**(const boost::container::flat_map<*Int*, *Int*> &f, *Val* &amp)

Maps the current distinguishability conf to a different one - Only use for mapping to less distinguishable conf.

Template Parameters
Val – Amplitude type that is used in the *State*

Parameters

- **f** – Encodes the mapping {old D.mode : new Dmode}
- **amp** – Amplitude for this key before the mapping

inline bool **notEmpty**(const std::vector<std::vector<*Int*>> &allModes) const

Checks if the key has at least one photon in on of the modes in allModes.

Parameters
allModes – Spatial&Distinguishability modes to check

Returns
true, if there is at least one photon in one of the mode

Returns
false, otherwise

inline bool **sameDModeDel**(const std::vector<*Int*> &m)

Checks if all S&P modes in m have the same distinguishability mode and deletes all modes in m.

Parameters
m – Spatial&Polarization modes to check

Returns
true, if all modes in m have the same distinguishability mode

Returns
false, otherwise

StateAux

Some helper functions for State.hpp.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Copyright
Copyright (c) 2024, provided under CC BY-NC 4.0. license

Functions

template<class **T**>
T **conj**(*T* t)

Performs the usual complex conjugate.

Template Parameters
T – Number type

Parameters
t – Number to perform the complex conj on.

Returns
T Complex conjugate

template<>
float **conj**<float>(float f)

Template specialization. Complex conjugate is identity for floats.

Parameters
t – Number to perform the complex conj on.

Returns
float Complex conjugate

template<>

double **conj**<double>(double f)

Template specialization. Complex conjugate is identity for doubles.

Parameters
t – Number to perform the complex conj on.

Returns
double Complex conjugate

template<>
int **conj**<int>(int f)

Template specialization. Complex conjugate is identity for ints.

Parameters
t – Number to perform the complex conj on.

Returns
int Complex conjugate

template<class **Val**, class **Real**>
Val **ovlpH**(const std::vector<*Val*> &b, const std::vector<*Real*> &wf, *Val* (*get_ovlp)(const std::vector<*Real*>&, const std::vector<*Real*>&), const std::vector<std::vector<*Real*>> &waves)

Computes the inner product of a OWF, and WF according to get_ovlp (cf. *State*).

Template Parameters

- **Val** – Value-type, cf. *State*.
- **Real** – Real-type, cf. *State*

Parameters

- **b** – Orthogonal wave function (OWF).
- **wf** – Wave function WF
- **get_ovlp** – Function that provides the overlap given two wave functions.
- **waves** – Wave functions that are used to express b. b provides the coefficients.

Returns
Val Overlap of b and wf.

template<class **Val**, class **Real**>
Val **ovlp**(const std::vector<*Val*> &b, const std::vector<*Val*> &c, *Val* (*get_ovlp)(const std::vector<*Real*>&, const std::vector<*Real*>&), const std::vector<std::vector<*Real*>> &waves)

Computes the inner product of two proto-orthogonal wave functions according to get_ovlp. Used in addBasisElem() (cf. *State*).

Template Parameters

- **Val** – Value-type, cf. *State*.
- **Real** – Real-type, cf. *State*.

Parameters

- **b** – proto-orthogonal wave function
- **c** – proto-orthogonal wave function
- **Function** – that provides the overlap given two wave functions.
- **waves** – Wave functions that are used to express b. b provides the coefficients.

Returns
Val Overlap of b and c.

State

Defines the central data structure *State*.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Copyright
Copyright (c) 2024, provided under CC BY-NC 4.0.

template<class **Key**, class **Val**, class **Real**>

class **State** : public boost::container::flat_map<*Key*, *Val*>

#include <*State.hpp*> Definition of the data structure used to represent states. Inherits from boost::container::flat_map<Key, Val>.

Template Parameters

- **Key** – *Key* type used in the *State*
- **Val** – Amplitude type used in the *State*, e.g. float or std::complex<float>
- **Real** – Real number type that should be used, e.g. float.

Public Functions

inline **State**()

Construct a new (empty) *State* object.

inline **State**(*Key* k)

Construct a new *State* object from a single key with amplitude 1.0.

Parameters
k – *Key* for the *State* (k:1.0).

inline void **set**(*Val* (*f)(const *WF*&, const *WF*&))

Sets the overlap function to f.

Parameters
f – function that gets two vector<Val> and returns a Val, that is the overlap.

inline void **set**(*Real* t)

Sets the tolerance to t.

Parameters
t – Real the tolerance is set to.

inline void **set**(*Int* n)

Sets the lossMode to n.

Parameters
n – Int the lossMode is set to.

inline void **set**(*Par* &&p) noexcept

Sets the actual *State* data to p.

Parameters
p – The data is set to. Moved input.

inline void **set**(const *Par* &p)

Sets the actual *State* data to p.

Parameters
p – The data is set to.

inline void **set**(const *Key* &k, const *Val* &v)

Inserts/assigns a new *Key* Value pair.

Parameters

- **k** – *Key* for new entry. If k is already in the *State* the amplitude is overwritten with v.
- **v** – new amplitude.

inline void **set**(*Key* &&k, const *Val* &v)

Inserts/assigns a new *Key* Value pair.

Parameters

- **k** – *Key* for new entry. If k is already in the *State* the amplitude is overwritten with v. *Key* is moved.
- **v** – new amplitude.

inline void **set**(*Int* a, *Int* b, *Int* c, const *Val* &v)

Inserts/assigns a new *Key* Value pair. The key is implicitly given by (a,b,c).

Parameters

- **a** – Spatial&Polarization mode
- **b** – Distinguishability mode
- **c** – Occupation number.
- **v** – new amplitude.

inline void **add**(const *Par* &p)

Adds an state-like object to a *State*. If a *Key* is already in the state the amplitudes are added.

Parameters
p – State-like object to be added.

inline void **add**(*Par* &&p)

Adds an state-like object to a *State*. If a *Key* is already in the state the amplitudes are added.

Parameters
p – State-like object to be added.

inline void **add**(const *Par* &p, const *Val* &v)

Adds an scaled state-like object to a *State*. If a *Key* is already in the state the amplitudes are added.

Parameters

- **p** – State-like object to be added.
- **v** – p is firsted scaled with v before added to this.

inline *Real* **norm**() const

Returns the norm of a state.

Returns
Real The norm of the state

inline *Par* **getPar**()

Get the Par, i.e. boost::container::flat_map<Key, Val>, object.

Returns
Par

inline *Par* &&**get_parMoved**()

Get the Par, i.e. boost::container::flat_map<Key, Val>, object moved.

Returns
Par&&

OWF **addBasisElem**(const *WF*&)

Uses Gram-Schmidt procedure to add a basis element to the orthogonal basis. Used to get orthogonal Distinguishability modes.

Parameters
wf – Wave-function of the new entry represented as vector

Returns
std::vector<Val> Returns the inner products of wf with all basis elements after G.-S. procedure.

void **addPhoton**(const *WF*&, *Int*, *Int*)

Adds num new photon to the state with the wave function wf in the Spatial&Polarization mode m.

Parameters

- **wf** – Wave function of the new photons represented as vector
- **mode** – Spatial&Polarization mode of the new photons
- **num** – Number of new photons

inline void **apply**(const std::vector<*Val*> &U, const std::vector<*Int*> &modes)

Applies a unitary U in second-quantization on the Spatial&Polarization modes in modes (max 2).

Parameters

- **U** – Unitary represented as line
- **modes** – modes affected by the operation

inline void **apply**(const *Val* &U, const *Int* &mode)

Applies a unitary U in second-quantization on the Spatial&Polarization modes in modes (max 1).

Parameters

- **U** – Unitary, i.e. here a number with abs value 1

- **mode** – mode affected by the operation

inline void **swap**(*Int* a, *Int* b)

Swaps two Spatial&Polarization modes, used to implement a PBS.

Parameters

- **a** – First Spatial&Polarization mode
- **b** – Second Spatial&Polarization mode

inline void **clean**()

Removes all Key-Val pairs where the abs of the amplitude is lower than tol.

inline void **normalise**()

Normalizes the state.

inline void **mul**(*Val*)

Multiplies the state with a scalar n.

Parameters

n – Scalar to multiply the state with

inline *State*<*Key*, *Val*, *Real*> **overlapWithFilter**(const boost::container::flat_map<*Int*, *Int*>&, const std::vector<std::vector<*Int*>>&, const std::vector<boost::container::flat_map<*Int*, *Int*>>&)

Computes the overlap to a measurement pattern and filters according to a fidelity reference.

Filters the state such that the measurement patter is overlapping and the fidelity reference is overlapping. Returns the part of the state that is accepted by the measurement and post-selection but is not overlapping with the fidelity refence.

Parameters

- **ref** – The measurement pattern {S&P mode: total occupation-numer}. All S&P modes not in ref are accepted either way.
- **allModes** – vector of vector of S&P modes of fidelity reference used to pre-filter. At least for one of them all modes have to be occupied. This implements post-selections.
- **fref** – The fidelity reference. Used to further filter the measurement result, keeps that parts that could contribute to the fidelity.

Returns

State<Key, Val, Real>

inline void **overlapCompl**(const std::vector<boost::container::flat_map<*Int*, *Int*>>&, const std::vector<std::vector<*Int*>>& = {})

Keeps the part of the state that is either rejected because of the measurement result or because of post-selection.

Parameters

- **MVec** – Measurement pattern {S&P mode: total occupation-numer}. Filters out the Keys that overlap with one of these.
- **allModes** – Modes for post-selection. Filters our those that are accepted by post-selection, i.e. for all of the vectors at least one is not occupied

inline void **loss**(const std::vector<*Int*>&)

Performes loss of one photons on the modes in modes.

Parameters
modes – Spatial&Polarization modes to perform loss on

inline void **collapse**(const *Key*&)

Maps the current distinguishability conf to a different one - Only use for mapping to less distinguishable conf.

Parameters
K – *Key* that encodes the target distinguishability configuration.

inline void **notEmpty**(const std::vector<std::vector<*Int*>>&)

Filters out these key that haven't at least one photon in on of the modes in allModes.

Parameters
allModes – Spatial&Distinguishability modes to check.

inline void **sameDModeDel**(const std::vector<std::vector<*Int*>>&, const std::vector<*Val*>&)

Keeps the keys where all Distinguishability Modes are the same for one of the collection of modes in modes.

Parameters

- **modes** – For one of the vectors in modes, all Distinguishability Modes have to be the same for the key to be included in the output.
- **facs** – Factors associated to the vectors in modes. These are used for the amplitudes correspoding to the keys, in which all the modes in the correspoding vector have the same Distinguishability Modes.

Private Types

using **Par** = boost::container::flat_map<*Key*, *Val*>

Short handle for the parent type that provides the actual data structure.

using **WF** = std::vector<*Real*>

Wave functions are parametrized as vectors of Reals. The overlap function needs to interprete these as intended by the user.

using **OWF** = std::vector<*Val*>

The orthogonal wave functions are represented are vectors of Vals, as they are amplitudes for the non-orthogonal wave functions after the Gram-Schmidt procedure which is implemented in *addBasisElem()*.

using **Int** = typename *Key*::basetype

Handle for the user integer number type from *Key*.

Private Members

std::vector<*WF*> **waves** = {}

The collections of non-orthogonal wave functions.

std::vector<*OWF*> **basis** = {}

The collections of orthogonal wave functions, build up using *addBasisElem()* and *addPhoton()*.

Val (***get_ovlp**)(const *WF*&, const *WF*&)

The function that returns the overlap given two WF. Can be set by the user using *set()*

Real **tol** = (*Real*)std::pow(10, -9)

The tolerance for the amplitudes. Amplitudes with lower absolute value are deleted in *clean()*.

Int **lossMode** = 0

Loss is modelled as a map to a new Spatial&Polarization mode. This mode should be always higher than modes used for computation.

SimAux

Some helper functions for the simulation.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Typedefs

template<class **T**>

using **CNum** = std::complex<*T*>

Functions

template<class **V**, class **R**>
std::array<std::vector<*V*>, 15> **genRotationsBasic**(const std::vector<*R*> &angleErrs)

Generate the rotation unitaries used for the wave-plates including the errors.

Template Parameters

- **V** – Value-type, cf. *State*
- **R** – Real-type, cf. *State*

Parameters

angleErrs – Rotation errors

Returns

std::array<std::vector<V>, 15> Unitaries for the rotations in line-form

template<class **R**>
inline void **write**(const std::vector<int> &LossPositions, const std::vector<int> &doublePrep, const std::vector<*R*> &angleErrs, const *R* &ovl, const std::string &path, int rank, const std::vector<*R*> &res)

Writes the results of the simulation to a file.

Template Parameters

R – Real number type that should be used, e.g. float.

Parameters

- **LossPositions** – Positions of loss in the circuit
- **doublePrep** – Spatial&Polarization modes with two-photon preparation
- **angleErrs** – Rotation error for wave plated
- **ovl** – pairwise overlap of wave functions
- **path** – Path-prefix where to save
- **rank** – Rank of the process, used for saving
- **res** – Result of the simulation

float **trivOvlF**(const std::vector<float> &V, const std::vector<float> &W)

Given to float-vectors that parametrize wave functions, it returs the overlap, which is trivially saved in the first vector.

Parameters

- **V** – First vector, at least len 2 with identifier on 0 and overlap on 1
- **W** – Second vector, at least len 1 with identifier on 0

Returns

float Overlap of the wave functions

template<class **K**, class **V**, class **R**>
void **detloss**(*State*<*K*, *V*, *R*> &S, int pos, const std::vector<int> &modes, const std::vector<int> &lossPos)

Applies loss on modes in S if the current position pos is in lossPos.

Template Parameters

- **K** – *Key* type that should be used.
- **V** – Amplitude type that should be used, e.g. float or std::complex<float>

- **R** – Real number type that should be used, e.g.

Parameters

- **S** – *State* to apply loss on
- **pos** – current position in circuit
- **modes** – modes affected by loss
- **lossPos** – potitions in circuit where loss happens

SimFid

Defines the function used for the simulation of GHZ state generation with imperfections.

Author
Fabian Wiesner (fabian.wiesner97@gmail.com)

Version
0.1

Date
2024-06-06

Functions

template<class **K**, class **V**, class **R**>
inline void **cleanOvlGHZ**(*State*<*K*, *V*, *R*> &S, int a)

Cleans input such that only the keys with the same DMode in one of the parts of the GHZ state remain.

Template Parameters

- **K** – Key-type, cf. *State*
- **V** – Value-type, cf. *State*
- **R** – Real-type, cf. *State*

Parameters

- **S** – *State* to compute to overlap for
- **a** – phase for GHZ state - either 1 or -1

inline void **fid**(const std::vector<std::array<*State*<*Key*<int>, float, float>, 8>> &PreData, const std::vector<std::array<*State*<*Key*<int>, float, float>, 8>> &Compl, const float &ovl, const std::vector<float> &angErrs, const std::vector<int> &doublePrep, const std::vector<int> &lossPos, const std::string &path, int rank)

Computes the fidelity for pre-computed data and writes it to a file.

Parameters

- **PreData** – Vector of Arrays (one for every input combination of DModes) of the remaining states after the measurement already projected onto the spatial and polarization modes of the GHZ state

- **Compl** – Same data structure as PreData. These contains the complement of the states after the measurement, i.e. those parts orthogonal to the GHZ state.
- **ovl** – Pairwise overlap
- **angErrs** – Angle errors in the setup
- **doublePrep** – Events/positions of double-preparation
- **lossPos** – Events/positions of loss
- **path** – The path to the folder where the data should be saves.
- **rank** – The rank of the current process. Used for saving the data to the file.

template<class **K**, class **V**, class **R**>
inline void **circuitFid**(*State*<*K*, *V*, *R*> &S, const std::vector<int> &lossPos, const std::array<std::vector<*V*>, 15> &apl)

The photonic circuit we considered to create a GHZ state.

Template Parameters

- **K** – Key-type, cf. *State*
- **V** – Value-type, cf. *State*
- **R** – Real-type, cf. *State*

Parameters

- **S** – *State* to perform the circuit on.
- **lossPos** – Positions where loss happens
- **apl** – Rotations as unitaries repr. as single line unitaries

void **collapseRenorm**(const *Key*<int> &K, std::array<*State*<*Key*<int>, float, float>, 8> &SVec, std::array<*State*<*Key*<int>, float, float>, 8> &comp, *State*<*Key*<int>, float, float> &S)

Maps a the perfectly distinguishable configuration to a partially distinguishability conf.

Parameters

- **K** – *Key* that encodes the desired distinguishability configuration
- **SVec** – States that should be used for the result, completly overlapping with GHZ in spatial and polarization and acceptable measurement results.
- **comp** – States that are used for normalization, i.e. part of the measurement result but orthogonal to GHZ, as map is not norm preserving
- **S** – Part of the state that is orthogonal to all acceptable measurement results

void **fidsim**(const std::vector<float> &ovls, const std::vector<int> &doublePrep, const std::vector<int> &lossPos, const std::vector<float> &angErrs, const std::array<std::vector<float>, 15> &apl, const std::string &path, int rank)

Computes the fidelity for a given parameters.

Parameters

- **ovls** – Overlaps, for all of them the fidelity is computed
- **doublePrep** – Spatial modes with two-photon preparation
- **lossPos** – Positions where loss happens
- **angErrs** – Rotation-errors for wave-plates

- **apl** – Rotations as unitaries repr. as single line unitaries (already including the rotation errors)
- **path** – Pathsuffix where to save the outcome
- **rank** – Rank of the process (used for saving the outcome)

void **schedulerGHZshuffled**(const std::vector<float> &ovls, std::vector<float> &angErrs, std::string path, int global_lower, int global_upper, int rank_off, int rank, int size, std::string shuffle_path)

This function iterates over most likely 10214 combinations of loss and two-photon creation and saves the fidelity and the probailities for all of them.

Parameters

- **ovls** – Overlaps, for all of them the fidelity is computed
- **angErrs** – Rotation-errors for wave-plates
- **path** – Pathsuffix where to save the outcome
- **global_lower** – Lower end of the intervall, between 0 and 10214
- **global_upper** – Upper end of the intervall, between 0 and 10214
- **rank_off** – Offset for the rank, which is used for saving the result
- **rank** – Rank of the process (used for saving the outcome)
- **size** – Number of processes
- **shuffle_path** – Path to a file where all 10214 combinations are shuffeled

INDEX

T

W