Documentation of Mermin-evaluation

Release 1.0

Henri de Boutray

Dec 19, 2019

CONTENTS

This is the reference manual for a SageMath package `grover_ent`.

To use this module, you need to import it:

```
from grover_ent import *
```

This package is used and documented in project available https://quantcert.github.io/Grover_ent

CHAPTER

ONE

GROVER ENTANGLEMENT

1.1 Grover main

This module is aimed to provide the user means to study the Grover algorithm and especially how the entanglement changes during its execution using the Mermin evaluation.

grover.**diffusion_artificial**(*V*)

Performs the inversion about the mean for V.

Parameters **V** (*vector*) – The running state in Grover's algorithm .

Returns vector – V Inverted about the mean.

grover.**grover**(*target_state_vector*, *verbose=False*, *file_name=None*, *use_precomputed=False*, *artificial=False*)

Prints in terminal or in file file_name the Mermin evaluation of each step of the Grover algorithm.

Example:

```
>>> grover(target_state_ket_string_to_vector("0000"))
[0.168628057515893, 1.32148634347176, 1.01189404012546, 0.469906068135870]
```

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **verbose** (*bool*) – If verbose then extra run information will be displayed in terminal.
- **file_name** (*str*) – File name for the registration of the Mermin evaluation for each step of the algorithm, in csv format.
- **use_precomputed** (*bool*) – for some states, the optimal Mermin polynomial has been precomputed, use this option to speed up the overall computation.
- **artificial** (*bool*) – Due to technological limits, it is not always possible to compute the states of Grover's algorithm in a realistic simulator. This parameters allows the user to use a non realistic simulator which is way quicker.

Returns any – Result of this function depends on file_name. If a file name is given qft_main returns None, otherwise, it returns an array with the evaluation values.

grover.**grover_artifical**(*target_state_vector*)

To accelerate the computation of the states, the alternative to grover_vanilla is to work directly on the vectors. INdeed, we know what the effects of each steps are, so we don't need to apply the gates we can simply change the vectors accordingly. For big dimensions, this is a way more efficient method.

Parameters **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).

Returns list[vector] – List of states after each application of the diffusion operator (as well as the initial state).

grover.**grover_evaluate**(*end_loop_states*, *M_opt*, *file_name*)

Uses the previously found optimal mermin operator to evaluate the entanglement for each state in `end_loop_states`.

Parameters

- **end_loop_states** (*list[vector]*) – The states at the end of each loop
- **M_opt** (*matrix*) – The optimal Mermin operator.
- **file_name** (*str*) – File name for the registration of the Mermin evaluation for each step of the algorithm, in csv format.

Returns any – Result of this function depends on file_name. If a file name is given `qft_main` returns None, otherwise, it returns an array with the evaluation values.

grover.**grover_layers_kopt**(*target_state_vector*)

Computes the circuit for Grover's algorithm using the circuit format used for the `run` function from `run_circuit.sage`.

Parameters **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).

Returns (list[list[matrix]],int) – Circuit for Grover's algorithm and optimal value found for `k_opt`.

grover.**grover_optimize**(*target_state_vector*, *use_precomputed=False*, *verbose=False*)

Computes the optimal Mermin operator to evaluate the entanglement in the Grover algorithm

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **use_precomputed** (*bool*) – For some states, the optimal Mermin polynomial has been precomputed, use this option to speed up the overall computation.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns matrix – The optimal Mermin operator.

grover.**grover_run**(*target_state_vector*, *artificial=False*, *verbose=False*)

Runs a simulation of the grover algorithm.

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **artificial** (*bool*) – due to technological limits, it is not always possible to compute the states of Grover's algorithm in a realistic simulator. This parameters allows the user to use a non realistic simulator which is way quicker.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns list[vectors] – the states at the end of each loop.

grover.**grover_vanilla**(*target_state_vector*, *verbose=False*)

Computes the successive states at the end of the loop using a realistic simulation of the execution of Grover's algorithm (using the simulator from `run_circuit.sage`).

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns list[vector] – List of states after each application of the diffusion operator (as well as the initial state).

grover.**oracle**(*target_state_vector*)

The oracle O satisfies $O|x, y >= |x, f(x) + y >$ where $f(x) = 1$ if x is the element looked for and $f(x) = 0$ otherwise.

Example: The oracle flips the qubit where the target is, so :

```
>>> oracle(target_state_ket_string_to_vector("101"))
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

Parameters **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).

Returns matrix – The matrix corresponding to the oracle.

grover.**oracle_artificial**(*target_state_vector*, *V*)

Flips the coefficient of `V` corresponding to the state `target_state_vector`.

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **V** (*vector*) – The running state in Grover's algorithm.

Returns vector – `V` with the correct coefficient flipped.

grover.**target_state_ket_list_to_vector**(*target_state_ket*)

Converts the target state from a digit list to a vector.

Example: $|5 >_4 = |0101 >= (0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$ so :

```
>>> target_state_ket_list_to_vector([0,1,0,1])
(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Parameters **target_state_ket** (*list[int]*) – List of digit of the boolean expression of the sate looked for.

Returns vector – Vector corresponding to the target state.

grover.**target_state_ket_string_to_vector**(*target_state_ket*)

Converts the target state from a string containing a digit list to a vector.

Example: `|5>_4 = |0101> = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)` so :

```
>>> target_state_ket_string_to_vector("0101")
(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Parameters **target_state_ket** (*str*) – List of digit of the boolean expression of the searched sate.

Returns vector – Vector corresponding to the target state.

grover.**target_state_ket_vector_to_string**(*target_state_ket*)

Converts the target state from a vector to a string containing a digit list.

Example: $|5>_4 = |0101> = (0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)$ so :

```
>>> v = vector((0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
>>> target_state_ket_vector_to_string(v)
'0101'
```

Parameters **target_state_ket** (*vector*) – Vector corresponding to the target state.

Returns string – List of digit of the boolean expression of the searched sate.

grover.**time_step**(*step_name*, *step_function*, *step_args*, *verbose=False*)

Times a function, with time information displayed if `verbose`.

Example:

```
>>> result = time_step("Add_10", lambda a : a + 10, [5], True)
Add_10 starts now
Add_10 complete, took 1.81198120117e-05s
>>> result
15
```

Parameters

- **setp_name** (*str*) – Name of the step, used for display.
- **step_function** (*function*) – Function to be timed.
- **step_args** (*tuple*) – Arguments of the function.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns any – The output of `step_function(*step_args)`.

1.2 QFT main

This module is aimed to provide the user means to study the QFT and especially how entanglement changes during its execution using the Mermin evaluation.

qft.**bit_value**(*n*, *k*, *base=2*)

Returns the value of the `k` th digit of the integer `n` given in base `base`.

Example:

```
>>> bit_value(5, 0)
1
>>> bit_value(5, 1)
0
>>> bit_value(15, 0, base=10)
5
```

Parameters

- **n** (*int*) – Integer studied.
- **k** (*int*) – Digit desired.
- **base** (*int*) – Base in which n is studied.

Returns int – Value of the k th digit of the integer n given in base base.

qft.**periodic_state**(*l*, *r*, *nWires*)

Returns the periodic state $|\varphi^{l,r}>$ of size 2^{nWires}. We have:

$|\varphi^{l,r}>=\sum_{i=0}^{A-1}|l+ir>/sqrt(A)$ with $A=floor((2^{nWires}-l)/r)+1$

In this definition, l is the shift of the state, and r is the period of the state.

Example: Since $|\varphi^{1,5}>=(|1>+|6>+|11>)/sqrt(3)=(|0001>+|0110>+|1011>)/sqrt(3)$,

```
>>> periodic_state(1,5,4)
(0, 1/3*sqrt(3), 0, 0, 0, 0, 1/3*sqrt(3), 0, 0, 0, 0, 1/3*sqrt(3), 0, 0, 0, 0)
```

Parameters

- **l** (*int*) – The shift of the state.
- **r** (*int*) – The period of the state.
- **nWires** (*int*) – The size of the system (number of qubits).

Returns vector – The state defined by l, r and nWires according to the definition given above.

qft.**qft_evaluate**(*states*, *verbose=False*, *file_name=None*)

Computes the Mermin Evaluation for each state in states.

Example:

```
>>> qft_evaluate([periodic_state(1,5,4)])
[1.33201398251237]
```

Parameters

- **states** (*list[vector]*) – The states after each step of the QFT.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.
- **file_name** (*str*) – File name for the registration of the Mermin evaluation for each step of the algorithm, in csv format.

Returns any – Result of this function depends on file_name. If a file name is given qft_main returns None, otherwise, it returns an array with the evaluation values.

qft.**qft_layers**(*state*)

Computes the circuit for the QFT using the circuit format used for the run function from run_circuit.sage.

Example:

```
>>> qft_layers(periodic_state(2,1,3))
[
  [
    [ 1/2*sqrt(2)  1/2*sqrt(2)]  [1 0]  [1 0]
    [ 1/2*sqrt(2) -1/2*sqrt(2)], [0 1], [0 1]
  ],[
    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 I 0]
    [0 0 0 0 0 0 0 I]
  ],[
    [1 0 0 0 0                     0 0                     0]
    [0 1 0 0 0                     0 0                     0]
    [0 0 1 0 0                     0 0                     0]
    [0 0 0 1 0                     0 0                     0]
    [0 0 0 0 1                     0 0                     0]
    [0 0 0 0 0 (1/2*I + 1/2)*sqrt(2) 0                     0]
    [0 0 0 0 0                     0 1                     0]
    [0 0 0 0 0                     0 0 (1/2*I + 1/2)*sqrt(2)]
  ],[
    [1 0]  [ 1/2*sqrt(2)  1/2*sqrt(2)]  [1 0]
    [0 1], [ 1/2*sqrt(2) -1/2*sqrt(2)], [0 1]
  ],[
    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 I 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 I]
  ],[
    [1 0]  [1 0]  [ 1/2*sqrt(2)  1/2*sqrt(2)]
    [0 1], [0 1], [ 1/2*sqrt(2) -1/2*sqrt(2)]
  ],[
    [1 0 0 0 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 0 0 0 1 0]
    [0 1 0 0 0 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 0 0 0 1]
  ]
]
```

Parameters **state** (*vector[int]*) – State on which the operator wants the QFT performed of, this is usually a periodic state.

Returns (list[list[matrix]],int) – Circuit for the QFT.

qft.**qft_main**(*state*, *verbose=False*, *file_name=None*)

Prints in terminal or in file `file_name` the Mermin evaluation of each step of the QFT.

Example:

```
>>> grover(periodic_state(0,11,4))0,1.99823485241887
1.99933058720672
1.99761683801972
```

(continues on next page)

(continued from previous page)

```
1.84600827724524
1.66149864543535
1.66010335826574
1.66127978203391
2.17163453049907
2.17187381801221
1.54230165695219
1.54129902140376
1.54169705834015
```

Parameters

- **state** (*vector[int]*) – State on which the operator wants the QFT performed of, this is usually a periodic state.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.
- **file_name** (*str*) – File name for the registration of the Mermin evaluation for each step of the algorithm, in csv format.

Returns any – Result of this function depends on file_name. If a file name is given qft_main returns None, otherwise, it returns an array with the evaluation values .

qft.**qft_matrix**(*size*)

This function should compute a matrix equivalent to the whole QFT operation. It has not been tester much though and should not be relied on for now.

Example:

```
>>> qft_matrix(3)
[   1/4*sqrt(2)    1/4*sqrt(2)    1/4*sqrt(2)    1/4*sqrt(2)    1/4*sqrt(2)    1/4*sqrt(2)    1/
↪4*sqrt(2)    1/4*sqrt(2)]
[   1/4*sqrt(2)    1/4*I + 1/4  1/4*I*sqrt(2)    1/4*I - 1/4   -1/4*sqrt(2)   -1/4*I - 1/4 -1/
↪4*I*sqrt(2)   -1/4*I + 1/4]
[   1/4*sqrt(2)  1/4*I*sqrt(2)   -1/4*sqrt(2) -1/4*I*sqrt(2)    1/4*sqrt(2)  1/4*I*sqrt(2)   -1/
↪4*sqrt(2) -1/4*I*sqrt(2)]
[   1/4*sqrt(2)    1/4*I - 1/4 -1/4*I*sqrt(2)    1/4*I + 1/4   -1/4*sqrt(2)   -1/4*I + 1/4  1/
↪4*I*sqrt(2)   -1/4*I - 1/4]
[   1/4*sqrt(2)   -1/4*sqrt(2)    1/4*sqrt(2)   -1/4*sqrt(2)    1/4*sqrt(2)   -1/4*sqrt(2)    1/
↪4*sqrt(2)   -1/4*sqrt(2)]
[   1/4*sqrt(2)   -1/4*I - 1/4  1/4*I*sqrt(2)   -1/4*I + 1/4   -1/4*sqrt(2)    1/4*I + 1/4 -1/
↪4*I*sqrt(2)    1/4*I - 1/4]
[   1/4*sqrt(2) -1/4*I*sqrt(2)   -1/4*sqrt(2)  1/4*I*sqrt(2)    1/4*sqrt(2) -1/4*I*sqrt(2)   -1/
↪4*sqrt(2)  1/4*I*sqrt(2)]
[   1/4*sqrt(2)   -1/4*I + 1/4 -1/4*I*sqrt(2)   -1/4*I - 1/4   -1/4*sqrt(2)    1/4*I - 1/4  1/
↪4*I*sqrt(2)    1/4*I + 1/4]
```

Parameters **size** (*int*) – The size of the QFT (number of qubits).

Returns matrix – The matrix corresponding to the QFT.

qft.**qft_run**(*state*, *verbose=False*)

Runs a simulation of the QFT.

Example:

```
>>> qft_run(periodic_state(2,1,3))
[(0, 0, 1/6*sqrt(6), 1/6*sqrt(6), 1/6*sqrt(6), 1/6*sqrt(6), 1/6*sqrt(6), 1/6*sqrt(6)),
 (1/12*sqrt(6)*sqrt(2), 1/12*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), -1/
↪12*sqrt(6)*sqrt(2), -1/12*sqrt(6)*sqrt(2), 0, 0),
 (1/12*sqrt(6)*sqrt(2), 1/12*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), -1/
↪12*sqrt(6)*sqrt(2), -1/12*sqrt(6)*sqrt(2), 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (1/12*sqrt(6)*sqrt(2), 1/12*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), 1/6*sqrt(6)*sqrt(2), -1/
↪12*sqrt(6)*sqrt(2), -(1/12*I + 1/12)*sqrt(6), 0, 0),
 (1/4*sqrt(6), 1/4*sqrt(6), -1/12*sqrt(6), -1/12*sqrt(6), -1/12*sqrt(6), -(1/24*I + 1/
↪24)*sqrt(6)*sqrt(2), -1/12*sqrt(6), -(1/24*I + 1/24)*sqrt(6)*sqrt(2)),
 (1/4*sqrt(6), 1/4*sqrt(6), -1/12*sqrt(6), -1/12*I*sqrt(6), -1/12*sqrt(6), -(1/24*I + 1/
↪24)*sqrt(6)*sqrt(2), -1/12*sqrt(6), -(1/24*I - 1/24)*sqrt(6)*sqrt(2)),
 (1/4*sqrt(6)*sqrt(2), 0, -(1/24*I + 1/24)*sqrt(6)*sqrt(2), (1/24*I - 1/24)*sqrt(6)*sqrt(2), -1/
↪24*sqrt(6)*sqrt(2) - (1/24*I + 1/24)*sqrt(6), -1/24*sqrt(6)*sqrt(2) + (1/24*I + 1/24)*sqrt(6), -1/
↪24*sqrt(6)*sqrt(2) - (1/24*I - 1/24)*sqrt(6), -1/24*sqrt(6)*sqrt(2) + (1/24*I - 1/24)*sqrt(6)),
 (1/4*sqrt(6)*sqrt(2), -1/24*sqrt(6)*sqrt(2) - (1/24*I + 1/24)*sqrt(6), -(1/24*I + 1/
↪24)*sqrt(6)*sqrt(2), -1/24*sqrt(6)*sqrt(2) - (1/24*I - 1/24)*sqrt(6), 0, -1/24*sqrt(6)*sqrt(2) +␣
↪(1/24*I + 1/24)*sqrt(6), (1/24*I - 1/24)*sqrt(6)*sqrt(2), -1/24*sqrt(6)*sqrt(2) + (1/24*I - 1/
↪24)*sqrt(6))]
```

Parameters

- **state** (*vector[int]*) – State on which the operator wants the QFT performed of, this is usually a periodic state.
- **verbose** (*bool*) – If `verbose` then extra run information will be displayed in terminal.

Returns list[vectors] – the list of states after each step.

`qft.`**`set_bit_value`**(*n*, *k*, *value*, *base=2*)

Returns `n` with its `k` th bit set to `value` in base `base`.

Example:

```
>>> set_bit_value(5, 0, 0)
4
>>> set_bit_value(5, 1, 0)
5
>>> set_bit_value(15, 0, 9, base=10)
19
```

Parameters

- **n** (*int*) – Integer modified.
- **k** (*int*) – Digit to change.
- **value** (*int*) – Value wanted for the `k` th digit of `n` (must be between 0 and `base-1`).
- **base** (*int*) – Base in which `n` is modified.

Returns int – `n` with its `k` th bit set to `value` in base `base`.

1.3 Run circuit

This module provides a simple quantum circuit simulator.

`run_circuit.`**`digit`**(*n*, *k*, *base=10*)

Computes the digit `k` for the integer `n` in its representation in base `base`.

Example:

```
>>> digit(152, 1)
5
>>> digit(152, 0)
2
```

Parameters

- **n** (*int*) – The integer for which the digit is needed.
- **k** (*int*) – The number of the digit.
- **base** (*int*) – The base used for the representation of n.

Returns int – The k th digit of n in base base.

run_circuit.**int_name**(*num*)

Converts a number to a string composed of the list of its digits in English.

Example:

```
>>> int_name(152)
'onefivetwo'
```

Parameters **num** (*int*) – Number to be converted to a list of digits.

Returns str – List of digits of num in base 10 concatenated.

run_circuit.**kronecker**(*a*, *b*)

Computes the Kronecker product of a and b.

Example:

```
>>> a = matrix([[1,0],[0,1]])
>>> b = matrix([[1,2],[3,4]])
>>> kronecker(a,b)
[1 2 0 0]
[3 4 0 0]
[0 0 1 2]
[0 0 3 4]
```

Parameters **a,b** (*matrix,vector*) – Operands for the kronecker operator.

Returns matrix *or* vector – The kronecker product of a and b (return type is the same a type of a).

run_circuit.**kronecker_power**(*a*, *n*)

Computes the n th Kronecker power of a.

Example:

```
>>> a = matrix([[1,0],[0,2]])
>>> kronecker_power(a,2)
[1 0 0 0]
[0 2 0 0]
[0 0 2 0]
[0 0 0 4]
```

Parameters

- **a** (*matrix,vector*) – The matrix (or vector) to be elevated to the n th power.
- **n** (*int*) – The power a has to be elevated to.

Returns matrix *or* vector – The n th kronecker power of a (return type is the same a type of a).

run_circuit.**layers_to_matrix**(*layers*)

layers is the same as matrix_layers but with matrix names embedded in them

Example:

```
>>> I2 = matrix.identity(2)
>>> I4 = matrix.identity(4)
>>> H = matrix([[1,1],[1,-1]])/sqrt(2)
>>> swap = matrix([[1,0,0,0],[0,0,1,0],[0,1,0,0],[0,0,0,1]])
>>> layers = [[('I',I4),('H',H)],[('H',H),('H',H),('I',I2)],[('I',I2),('S',swap)]]
>>> layers_to_matrix(layers)
[[I4,H],[H,H,I2],[I2,swap]]
```

Parameters **layers** (*list[list[(str,matrix)]]*) – Circuit under the format described above.

Returns list[List[(str,int)]] – Circuit under the ready-to-run format described above.

run_circuit.**layers_to_printable**(*layers*)

layers is the same as matrix_layers but with matrix names embedded in them

Example:

```
>>> I2 = matrix.identity(2)
>>> I4 = matrix.identity(4)
>>> H = matrix([[1,1],[1,-1]])/sqrt(2)
>>> swap = matrix([[1,0,0,0],[0,0,1,0],[0,1,0,0],[0,0,0,1]])
>>> layers = [[('I',I4),('H',H)],[('H',H),('H',H),('I',I2)],[('I',I2),('S',swap)]]
[[('I',2),('H',1)],[('H',1),('H',1),('I',1)],[('I',1),('S',2)]]
```

Parameters **layers** (*list[list[(str,matrix)]]*) – Circuit under the format described above.

Returns list[List[(str,int)]] – Circuit under the ready-to-print format described above.

run_circuit.**output_commant**(*command_name*, *command*, *output=False*, *output_to_file=False*, *w_file=None*)

This function is used to print useful informations in various ways, see arguments details for more information.

Example:

```
>>> output_commant("test",vector(SR,[1,0,0,1]), output=True)
\newcommand{\test}{1 \ket{00} + 1 \ket{11}}
```

Parameters

- **command_name** (*str*) – The command name.
- **command** (*any*) – The command content (if type is `str`, will be printed as such; if `vector`, `vector_to_ket` will be called on it and if `matrix`, `latex` method from SageMath will be called on it).
- **output** (*bool*) – disables or enables the output.
- **output_to_file** (*bool*) – Whether the output should be in the standard output or in a file.
- **w_file** (*file*) – The opened file to write the output to.

Returns None

run_circuit.**print_circuit**(*name_layers*, *to_latex=False*)

Each name is a tuple with the name of the gate and its dimension

Example:

```
>>> circuit = [[('I',2),('H',1)],[('H',1),('H',1),('I',1)],[('I',1),('S',2)]]
>>> print_circuit(circuit, to_latex=False)
---H---
---H-S-
-H---|-
>>> print_circuit(circuit, to_latex=True)
\begin{align*}
 \Qcircuit @C=1em @R=.7em {
  & \qw       & \gate{H}  & \qw               & \qw\\
  & \qw       & \gate{H}  & \multigate{1}{S}  & \qw\\
  & \gate{H}  & \qw       & \ghost{S}         & \qw
 }
\end{align*}
```

`@multi-gost_`, `@multi-source_` and `@multi-size_` are reserved names, they should not be used as a gate name.

Parameters

- **name_layers** (*list[List[(str,int)]]*) – Circuit name formalism in the shape of the example given above. First layer should not be empty. sum([item[1] for item in layer]) should be constant for layer in layers
- **latex** (*bool*) – If true, a string is returned containing the circuit in the format given by the LaTeX package `qcircuit`. Otherwise, the string returned is under a custom format meant to be easily readable in the terminal.

Returns str – The circuit in the format described above.

run_circuit.**run**(*matrix_layers*, *V_init*, *output=False*, *output_file=False*, *file=None*, *vector_name='V'*, *matrix_name='M'*)

Runs the algorithm specified by the matrix_layers.

Example:

```
>>> I2 = matrix.identity(2)
>>> I4 = matrix.identity(4)
>>> H = matrix([[1,1],[1,-1]])/sqrt(2)
>>> swap = matrix([[1,0,0,0],[0,0,1,0],[0,1,0,0],[0,0,0,1]])
>>> v = vector([1, 0, 0, 0, 0, 0, 0, 0])
>>> layers = [[I4,H],[H,H,I2],[I2,swap]]
>>> run(layers,v)
([(1, 0, 0, 0, 0, 0, 0, 0),
  (1/2*sqrt(2), 1/2*sqrt(2), 0, 0, 0, 0, 0, 0),
  (1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/
↪4*sqrt(2)),
  (1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/4*sqrt(2), 1/
↪4*sqrt(2))],
 [
[ 1/2*sqrt(2)  1/2*sqrt(2)            0            0            0            0            0        ␣
↪    0]
[ 1/2*sqrt(2) -1/2*sqrt(2)            0            0            0            0            0        ␣
↪    0]
[            0            0  1/2*sqrt(2)  1/2*sqrt(2)            0            0            0        ␣
↪    0]
[            0            0  1/2*sqrt(2) -1/2*sqrt(2)            0            0            0        ␣
↪    0]
[            0            0            0            0  1/2*sqrt(2)  1/2*sqrt(2)            0        ␣
↪    0]
[            0            0            0            0  1/2*sqrt(2) -1/2*sqrt(2)            0        ␣
↪    0]
[            0            0            0            0            0            0  1/2*sqrt(2)  1/
↪2*sqrt(2)]
[            0            0            0            0            0            0  1/2*sqrt(2) -1/
↪2*sqrt(2)],
```

(continues on next page)

(continued from previous page)

```
[ 1/2    0  1/2    0  1/2    0  1/2    0]  [1 0 0 0 0 0 0 0]
[   0  1/2    0  1/2    0  1/2    0  1/2]  [0 0 1 0 0 0 0 0]
[ 1/2    0 -1/2    0  1/2    0 -1/2    0]  [0 1 0 0 0 0 0 0]
[   0  1/2    0 -1/2    0  1/2    0 -1/2]  [0 0 0 1 0 0 0 0]
[ 1/2    0  1/2    0 -1/2    0 -1/2    0]  [0 0 0 0 1 0 0 0]
[   0  1/2    0  1/2    0 -1/2    0 -1/2]  [0 0 0 0 0 0 1 0]
[ 1/2    0 -1/2    0 -1/2    0  1/2    0]  [0 0 0 0 0 1 0 0]
[   0  1/2    0 -1/2    0 -1/2    0  1/2], [0 0 0 0 0 0 0 1]
])
```

Parameters

- **matrix_layers** (*array[array[matrix]]*) – Algorithm described as it would be in a circuit.
- **V_init** (*vector*) – Initial input of the algorithm.
- **output** (*bool*) – If true, outputs are enabled.
- **output_to_file** (*bool*) – If true, trace is returned in file, else it is printed.
- **file** (*file*) – File to output the commands to.
- **vector_name** (*str*) – Base name used for the vectors commands.
- **matrix_name** (*str*) – Base name used for the matrices commands.

Returns vector – The states along the execution of the algorithm as well as the matrix corresponding to each layer.

1.4 Optimization

This module is an simple optimization module that helps find the point of maximum value for a given function.

opti.**optimize**(*func*, *args_init*, *step_init*, *step_min*, *iter_max*, *verbose=False*)

Optimization function finding the maximum reaching coordinates for `func` with a random walk.

Parameters

- **func** (*function*) – The function to be optimized, each of its arguments must be numerical and will be tweaked to find `func`'s maximum.
- **args_init** (*tuple*) – Initial coordinates for the random walk.
- **step_init** (*real*) – The step size will vary with time in this function, so this is the initial value for the step size.
- **step_min** (*real*) – The limit size for the step.
- **iter_max** (*int*) – Upper iterations bound for each loop to avoid infinite loops.
- **verbose** (*bool*) – If `verbose` then extra run information will be displayed in terminal.

Returns vector, complex – The coordinates of the optimum found for `func` and the value of `func` at this point.

opti.**optimize_2spheres**(*func*, *args_init*, *step_init*, *step_min*, *iter_max*, *radius=1*, *verbose=False*)

Optimization function finding the maximum reaching coordinates for `func` with a random walk on a two sphere of dimension half the size of `args_init`. (Work in progress !)

For now, this function is in project and is not used, it can be ignored.

Parameters

- **func** (*function*) – The function to be optimized, each of its arguments must be numerical and will be tweaked to find func's maximum.
- **args_init** (*tuple*) – Initial coordinates for the random walk.
- **step_init** (*real*) – The step size will vary with time in this function, so this is the initial value for the step size.
- **step_min** (*real*) – The limit size for the step.
- **iter_max** (*int*) – Upper iterations bound for each loop to avoid infinite loops.
- **radius** (*real*) – Sphere radius.
- **verbose** (*bool*) – If verbose then extra run information will be displayed in terminal.

Returns vector, complex – The coordianates of the optimum found for func and the value of func at this point.

opti.**optimize_normalized**(*func*, *normalizer_func*, *args_init*, *step_init*, *step_min*, *iter_max*, *verbose=False*)

Optimization function finding the maximum reaching coordinates for func with a random walk.

Parameters

- **func** (*function*) – The function to be optimized, each of its arguments must be numerical and will be tweaked to find func's maximum.
- **args_init** (*tuple*) – Initial coordinates for the random walk.
- **step_init** (*real*) – The step size will vary with time in this function, so this is the initial value for the step size.
- **step_min** (*real*) – The limit size for the step.
- **iter_max** (*int*) – Upper iterations bound for each loop to avoid infinite loops.
- **verbose** (*bool*) – If verbose then extra run information will be displayed in terminal.

Returns vector, complex – The coordinates of the optimum found for func and the value of func at this point.

1.5 Mermin evaluation

This module is a SageMath module aimed at computing Mermin operators optimized to detect a specific quantum state.

mermin_eval.**M**(*n*, *a*, *a_prime*)

M_n is defined as such: M_n = (1/2)*(M_(n-1).tensor(a + a') + M'_(n-1).tensor(a - a'))

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **a,a_prime** (*matrix*) – Size 2 hermitian operators, defining M as given above.

Returns matrix – A size 2^n operator, following the definition given above.

mermin_eval.**M_all**(*_n*, *_a*, *_a_prime*)

M_n is defined as such: M_n = (1/2)*(M_(n-1).tensor(a_n + a_n') + M'_(n-1).tensor(a-n - a-n'))

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **a,a_prime** (*list[matrix]*) – List of size 2 hermitian operators, defining M as given above.

Returns matrix – A size 2^n operator, following the definition given above.

mermin_eval.**M_eval**(*a*, *b*, *c*, *m*, *p*, *q*, *phi*)

This function evaluates <*phi*|*M_n*|*phi*> with *(a,b,c,m,p,q)* describing M_n, the Mermin operator.

M_n traditionally uses two families of operators, a_n and a'_n, in our case, a_n = a*X+b*Y+c*Z and a'_n = m*X+p*Y+q*Z.

Parameters

- **a,b,c,m,p,q** (*real*) – Coefficients for the Mermin operator, used as described above.
- **phi** (*vector[complex]*) – Vector to be evaluated with M.

Returns complex – <*phi*|*M_n*|*phi*>

mermin_eval.**M_eval_all**(*_n*, *_a_coefs*, *_a_prime_coefs*, *_rho*)

This function evaluates tr(M_n * rho) with *a* and *a'* describing Mn, the Mermin operator.

The coefficients must be given in the following shape: >>> [[a,b,c], [d,e,f], ...] and will result in the following family of observables: >>> a[0] = a*X + b*Y + c*Z >>> a[1] = d*X + e*Y + f*Z >>> ...

Parameters

- **_n** (*int*) – Size of the system.
- **_a_coefs, _a_prime_coefs** (*list[list[real]]*) – Coefficients for the Mermin operator, used as described above.
- **rho** (*matrix[complex]*) – Density matrix of the state to be evaluated with M_n

Returns complex

mermin_eval.**M_from_coef**(*n*, *a*, *b*, *c*, *m*, *p*, *q*)

Returns the Mermin operator for a given size *n* and coefficients *a* through *q*.

M traditionally uses two families of operators, a_n and a'_n, in our case, a_n = a*X+b*Y+c*Z and a'_n = m*X+p*Y+q*Z.

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **a,b,c,m,p,q** (*real*) – Coefficients for the Mermin operator, used as described above.

Returns matrix – The Mermin operator M_n.

mermin_eval.**M_from_coef_all**(*_n*, *_a_coefs*, *_a_prime_coefs*)

Returns the Mermin operator for a given size *n* and coefficients of *a* and *a'*

The coefficients must be given in the following shape: >>> [[a,b,c], [d,e,f], ...] and will result in the following family of observables: >>> a[0] = a*X + b*Y + c*Z >>> a[1] = d*X + e*Y + f*Z >>> ...

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **_a_coefs, _a_prime_coefs** (*list[list[real]]*) – Coefficients for the Mermin operator, used as described above.

Returns matrix – The Mermin operator M_n.

mermin_eval.**M_prime**(*n*, *a*, *a_prime*)

M'_n is defined as such: M'_n = (1/2)*(M'_(n-1).tensor(a + a') + M_(n-1).tensor(a' - a))

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **a,a_prime** (*matrix*) – Size 2 hermitian operators, defining M as given above.

Returns matrix – A size 2^n operator, following the definition given above.

mermin_eval.**M_prime_all**(*_n*, *_a*, *_a_prime*)

M'_n is defined as such: M'_n = (1/2)*(M'_(n-1).tensor(a_n + a_n') + M_(n-1).tensor(a_n' - a_n))

Parameters

- **n** (*int*) – Iteration for the Mermin operator (determines its size).
- **a,a_prime** (*list[matrix]*) – List of size 2 hermitian operators, defining M' as given above.

Returns matrix – A size 2^n operator, following the definition given above.

mermin_eval.**coefficients_packing**(*_a_a_prime_coefs*)

Packs a list of elements in two lists of lists of three elements

Example: >>> coefficients_packing([1,2,3,4,5,6,7,8,9,10,11,12]) ([[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]])

This function is used to interface above ... *_all* functions and the *optimize* function.

Parameters **_a_a_prime_coefs** (*list[any]*) – List of elements.

Returns tuple[list[list[any]]] – Lists of lists of elements as described above.

mermin_eval.**coefficients_unpacking**(*_a_coefs*, *_a_prime_coefs*)

Unpacks two lists of lists of three elements to one list of elements

Example: >>> coefficients_unpacking([[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]) [1,2,3,4,5,6,7,8,9,10,11,12]

This function is used to interface above ... *_all* functions and the *optimize* function.

Parameters **_a_coefs, _a_prime_coefs** (*tuple[list[list[any]]]*) – Lists of lists of elements as described above.

Returns list[any] – List of elements.

mermin_eval.**mermin_coef_opti**(*target_state*, *verbose=False*)

Returns the Mermin operator maximizing the measure for a given input.

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns matrix, real – Coefficients of the optimal Mermin operator for *target_state* in th Grover algorithm and the value reached.

mermin_eval.**mermin_coef_opti_all**(*phi*, *verbose=False*)

Returns the Mermin operator' coefficients maximizing tr(M_n * rho) for a given input phi (where $rho = |phi >< phi|$ is the density matrix corresponding to the state phi).

Parameters

- **phi** (*vector[complex]*) – State used for the optimization of tr(M_n * rho) (only single item searches are supported for now).
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns list[real], real – Coefficients of the optimal Mermin operator for *target_state* in th Grover algorithm and the value reached.

mermin_eval.**mermin_operator_opti**(*target_state_vector*, *precomputed_filename=None*, *verbose=False*)

Computes the pseudo optimal operator used to perform the Mermin evaluation during Grover's algorithm.

Parameters

- **target_state_vector** (*vector[int]*) – State searched by Grover's algorithm (only single item searches are supported for now).
- **precomputed_filename** (*str*) – File where the precomputed coefficients for optimal Mermin operator are stored, if left empty, precomputation will not be used. If precomputation is used and the searched state is not in this database, once the optimization done, the result will be added to the file.
- **verbose** (*bool*) – If *verbose* then extra run information will be displayed in terminal.

Returns matrix, real – The Mermin operator satisfying the required conditions and the value reached.

1.6 Vector to Ket

This module is an ease of life adding to SageMath: it complements the `latex` method to quantum ket notations.

vector_to_ket.**int_index_to_ket**(*index*, *register_size*)

Makes a int from a register into a binary ket notation. IMPORTANT: this notation requires the *physics* package in Latex

Example:

```
>>> int_index_to_ket(2,2)
'\\ket{10}'
```

Parameters

- **index** (*int*) – the index to be transformed
- **register_size** (*int*) – the number of qubits in the system

Returns string – a string in latex format

vector_to_ket.**vector_to_ket**(*v*)

Transform a sage vector to a Latex ket notation

Example:

```
>>> v = vector(SR,[1,0,0,1])
>>> vector_to_ket(v)
1 \ket{00} + 1 \ket{11}
```

Parameters **v** (*vector*) – the vector corresponding to a pure state (size must be a power of two)

Returns str – a string corresponding to the Latex code for the ket notation of the input vector

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

O

P

Q

R

S

T

V