commit01

April 7, 2022

```
[ ]: from lib.functions0 import *
import numpy as np
import datetime
from qiskit.ignis.mitigation.measurement import CompleteMeasFitter
from qiskit import Aer, assemble, QuantumCircuit, QuantumRegister,␣
 ↪ClassicalRegister, IBMQ, transpile, execute
from qiskit.providers.aer import AerSimulator, QasmSimulator
from qiskit.opflow import Zero, One, I, X, Y, Z
from qiskit.ignis.verification.tomography import state_tomography_circuits,␣
 ↪StateTomographyFitter
from qiskit.quantum_info import state_fidelity
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
import warnings
warnings.filterwarnings('ignore')
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q-community',
                            group='ibmquantumawards',
                            project='open-science-22')

backend_sim_jakarta = QasmSimulator.from_backend(provider.
 ↪get_backend('ibmq_jakarta'))
backend_real_jakarta = provider.get_backend('ibmq_jakarta')
backend_sim = Aer.get_backend('qasm_simulator')
```

0.1 IBM open-science-prize-2021/22 solution. By Quantum Polo Gang: Ruben, Fabio & Valerio

0.2 Decomposition:

- We computed numerically the operator of N trotter steps, for a certain evolution time: U^n
- Observing that this operator preserves the magnetization of the system, if the initial state belongs to an eigenspace of the magnetization is possible to decompose the operator with 4 c-not. If the initial state is a superposition of states with different magnetization the best decomposition we found has 11 c-not (14 for the Jakarta geometry).
- Our initial state is $|110>$ (qubits 5,3 and 1 respectively) so we can use the best decomposition (4 c-not).

To see the decomposition procedure open *decomposition.ipynb* file.

Let's start from the defining of the evolution circuit parameters:

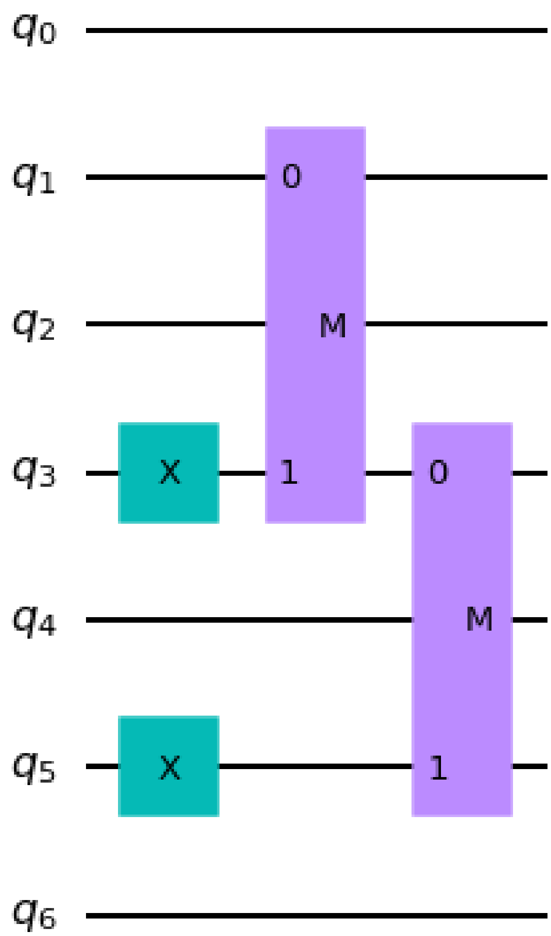
- *steps*: number of trotter steps (integer).
- *time*: time of evolution (double).
- *initial_state*: the 3-qubit initial state (string): from right to left, associated with qubits 1, 3 and 5 respectively
- *reps*: number of times each circuit is runned, in order to compute a standard deviation of the fidelity.
- *shots*: number of shots for every run.
- *backend*: here you can choose on which backend run the simulation: *backend_sim_jakarta* (noisy simulator), *backend_real_jakarta* (real device), *backend_sim* (simulator)

[ ]:
```
n_steps=12
time=np.pi
initial_state={"110": 1}
reps = 1
shots = 80
backend = backend_sim_jakarta

qr, qc = evolution_cirquit_single_state( time=time, n_steps=n_steps,␣
 ↪initial_state=initial_state) #DEVO IMPLEMENTARLA PER QUALSIASI STATO␣
 ↪INIZIALE!
qc.draw(output="mpl")
```

```
metti apposto lo stato iniziale!
6
2
```

[ ]:



Then we add the copy check for the mitigation (see *Ancillas_Error_mitigation_Git_Hub.pynb*)

[ ]:
```
qc = add_symmetry_check(qc, [qr[1],qr[3],qr[5]], [qr[0],qr[2],qr[4],qr[6]],␣
 ↪type="4copy_check")
qc.draw(output="mpl")
```

Then, we build the tomography circuits

[ ]:
```
from qiskit.ignis.verification.tomography import state_tomography_circuits,␣
 ↪StateTomographyFitter

qcs = state_tomography_circuits(qc, [qr[1],qr[3],qr[5]])
qcs_na = state_tomography_circuits(qc, [qr[1],qr[3],qr[5]])
```

```
## add the measure on the ancillas

for qc in qcs:
    cr_anc = ClassicalRegister(4)
    qc.add_register(cr_anc)
    #qc.barrier()
    qc.measure([0,2,4,6],cr_anc)

## qcs_tot is a list holding the tomography cirquits reps times.

qcs_tot = []
for _ in range(reps):
    qcs_tot=qcs_tot + qcs

qcs[10].draw(output="mpl")
```

Building the calibration circuits

```
[ ]: qcs_calibs, meas_calibs = calibration_cirquits("column_evolution_remake",␣
     ↪q_anc=[0,2,4,6], check="yes", check_type="4copy_check")
     state_lables = bin_list(7)
```

```
[ ]: state_lables[0]
```

```
[ ]: meas_calibs[2].draw(output="mpl")
```

```
[ ]: qcs_calibs[2].draw(output="mpl")
```

Than we run all the circuits

```
[ ]: jobs_evo=execute(qcs_tot, backend=backend, shots=shots)
     job_cal_our=execute(qcs_calibs, backend=backend, shots=shots)
     job_cal=execute(meas_calibs, backend=backend, shots=shots)
```

```
[ ]: jobs_evo_result = jobs_result(job_evolution = jobs_evo, reps = reps,␣
     ↪ancillas=[0,2,4,6])
```

or we can retrieve the jobs

```
[ ]: #evo_ID = "6233ae39d97bff04d66929e9"
     #cal_ID = "6233ae3ba2f72dff43da994f"

     #evo_job=backend.retrieve_job(evo_ID)
     #job_cal_our=backend.retrieve_job(cal_ID)
     '''
     reps=8
     steps=42
     backend=backend_real_jakarta
```

```
job_cal_our =backend.retrieve_job("6237aee18293e9eb4e1e4c4a")
job_cal =backend.retrieve_job("6237aedf0af65dc88cd92302")

job=backend.jobs(limit=30, start_datetime= "2022-03-19",␣
↪end_datetime="2022-03-26")[2]

jobs_evo_result = jobs_result(job_evolution = job, reps = reps,␣
↪ancillas=[0,2,4,6])
'''
```

[ ]:
```
#### DA CANCELLAREEEEEEEEEEEEEEEEEEEE
#state_lables = bin_list(7)
#qcs_na = circuits_without_ancillas_measuraments(job)
```

Next we apply the mitigation in the following way:

- measure mitigation: we apply the inverse of the calibration matrix to each circuit (see *measure_mitigation.ipynb*)
- ancillas mitigation: we throw away all the measures which contain a value for the ancillas physically forbidden.

this is done by the *mitigate* function.

Then we compute the fidelity for both the mitigated results and not-mitigated ones, in order check the gain given by the mitigation.

[ ]:
```
meas_fitter_our = CompleteMeasFitter(job_cal_our.result(),␣
↪state_labels=state_lables)
meas_fitter = CompleteMeasFitter(job_cal.result(), state_labels=state_lables)



target_state = (One^One^Zero).to_matrix()

fids=np.zeros([reps,4])
fids_mean=np.zeros(4)
fids_dev=np.zeros(4)

for j in range(reps):

    res = jobs_evo_result[j]
    print(j)
    new_res, new_res_nm = mitigate(res, Measure_Mitig="yes",␣
↪ancillas_conditions=['0011','1110','1101'], meas_fitter=meas_fitter)
    new_res_our, new_res_nm = mitigate(res, Measure_Mitig="yes",␣
↪ancillas_conditions=['0011','1110','1101'], meas_fitter=meas_fitter_our)
    new_res_not_mitigated = mitigate(res, Measure_Mitig="no",␣
↪ancillas_conditions=bin_list(4))
```

```
    fids[j,0] = fidelity_count(new_res_not_mitigated, qcs_na, target_state)
    fids[j,1] = fidelity_count(new_res_nm, qcs_na, target_state)
    fids[j,2] = fidelity_count(new_res, qcs_na, target_state)
    fids[j,3] = fidelity_count(new_res_our, qcs_na, target_state)

for i in range(4):
    fids_mean[i]=np.mean(fids[:,i])
    fids_dev[i]=np.std(fids[:,i])
```

[ ]:
```
new_res_our.get_counts(-1)
```

Printing the fidelity

[ ]:
```
labels = ["raw:                                      ",
          "ancillas mitigation:                      ",
          "ancillas and qiskit measurement mitigation:",
          "ancillas and our measurement mitigation:   "
]

for i in range(4):
    print(labels[i], fids_mean[i], " +- ", fids_dev[i])
```

[ ]:
```

```