Randomized Benchmarking

A key requirement for scalable quantum computing is that elementary quantum gates can be implemented with sufficiently low error. One method for determining the error behaviour of a gate implementation is to perform process tomography. However, standard process tomography is limited by errors in state preparation, measurement and one-qubit gates. It suffers from inefficient scaling with number of qubits and does not detect adverse error compounding when gates are composed in long sequences. An additional problem is due to the fact that desirable error probabilities for scalable quantum computing are of the order of 0.0001 or lower. Experimentally proving such low errors is challenging. We describe a randomized benchmarking method that yields estimates of the computationally relevant errors without relying on accurate state preparation and measurement. Since it involves long sequences of randomly chosen gates, it also verifies that error behaviour is stable when used in long computations. We implemented randomized benchmarking on 2-qubit simultaneous with 1-qubit, establishing a **Error per Clifford (EPC) 1.320763e-02**. In this particular experiment, we have 3 qubits Q0,Q1,Q2. We are running 2Q RB (on qubits Q0,Q2) and 1Q RB (on qubit Q1) simultaneously, where there are twice as many 1Q Clifford gates. We define a noise model for the simulator. To simulate decay, we add depolarizing error probabilities to the CNOT and U gates. We execute these sequences, but with a noise model extended with T1/T2 thermal relaxation error, and fit the exponentially decaying curve. We count the number of **gates per Clifford**, and calculate the **two-qubit Clifford gate error**, using the predicted primitive gate errors from the coherence limit.

What is Randomized Benchmarking?

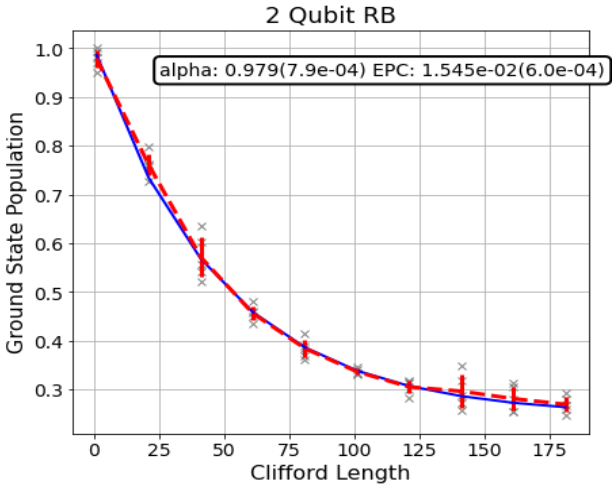
- A proven protocol that provides an efficient and reliable estimate of an average error-rate for a set of quantum gate operations.

Consists of the following three stages:

a) Generate RB sequences consisting of random elements from a Clifford group, including a computed reversal gate

b) Run the RB sequences either on the device or on a simulator (with a noise model) and compare to the initial state

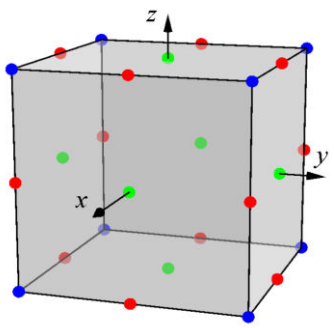
c) Get the statistics and fit an exponential decaying curve: Ap^m+B Error per Clifford (EPC): $r=(1-p)(d-1)/d \quad (d=2n)$



Clifford Group :

- The **Clifford group** consists of the quantum operators that can be efficiently simulated (in polynomial time) using a classical computer –Clifford simulation.
- It is generated by the gates: *H*, *S*and *CNOT:*

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \qquad S = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}, \qquad CNOT = \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

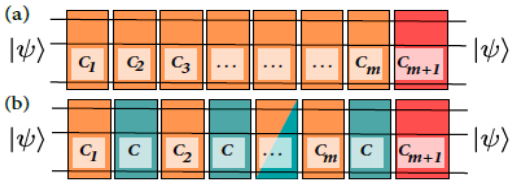


- The Clifford group on 1-qubit has 24 elements (the rotational symmetries of the cube)
- The Clifford group on n-qubit is of size: $2^{n^2+2n}\prod_{j=1}^{n}(4^j-1)$
- Efficient algorithms to generate and synthesize Clifford elements
- The Clifford group forms a *unitary 2-design–twirling* the finite Clifford group is equivalent to *twirling* the infinite unitary group

RB methods for estimating noise parameters :

- **Interleaved RB** Estimating the average error of individual quantum gates

$$r_C^{\text{est}} = \frac{(d-1)\,(1-p_{\overline{C}}/p)}{d},$$



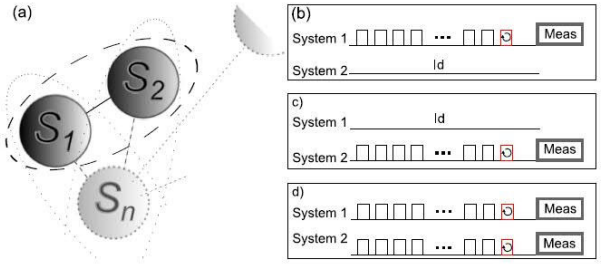
- **Purity RB :** Quantifies how coherent the errors are, namely

$$Tr(\rho^2) = 1 \;\leftrightarrow\; \rho \; is \; pure \; (coherent)$$

- **Leakage RB :** Quantification and characterization of leakage errors (unwanted energy levels)

- **Simultaneous RB :** Running RB simultaneously on subsets of qubits, characterizing the amount of addressability between subsystems

- **Correlated RB :** Estimating correlated crosstalk error between qubits participating in separate gates



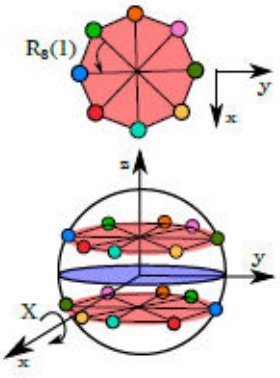
RB methods for other groups and gates

- **Non-Clifford Dihedral and CNOT-Dihedral RB**

$$D_m = < X, Z_m >$$

$$Z_m = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/m} \end{bmatrix}$$

$$G_m = < CX(i,j), X(j), Z_m(j) >$$



- Allows benchmarking of non-Clifford gates: T, CS, CCZ
- Efficient synthesis of CNOT-Dihedral elements

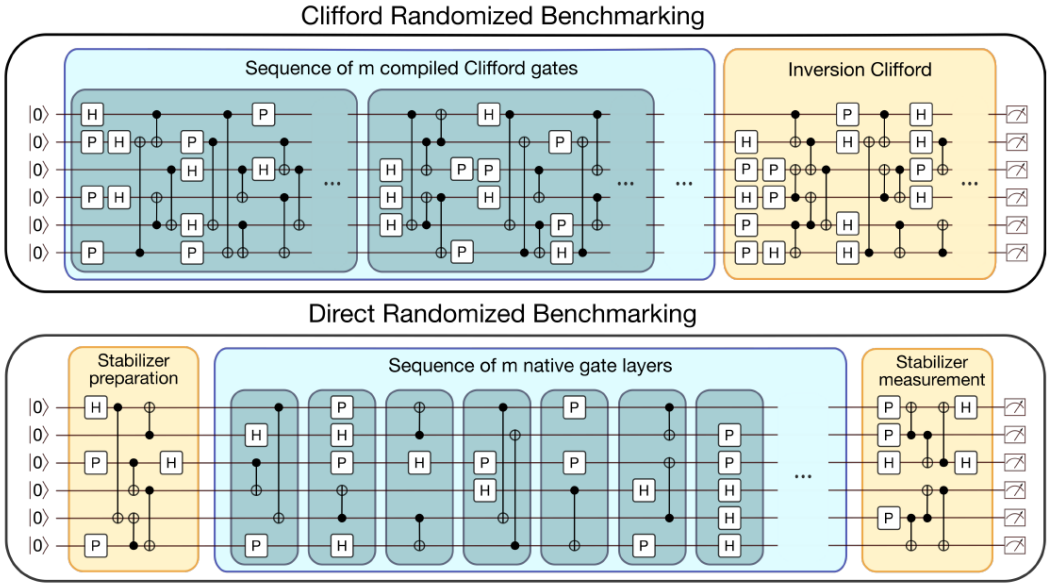
- **Pauli and CNOT-Pauli RB :**

The Pauli group on n-qubits is generated by the tensor product of the Pauli matrices

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- Subgroups of the Clifford group
- More efficient synthesis compared to the Clifford group

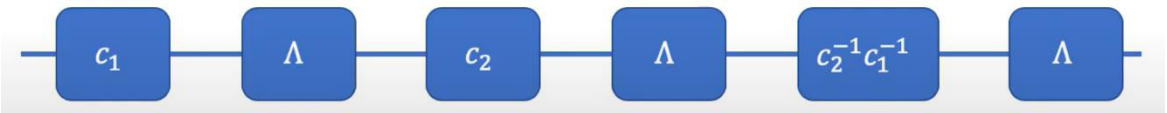
- **Direct RB :** Directly benchmarking the native gates of a device



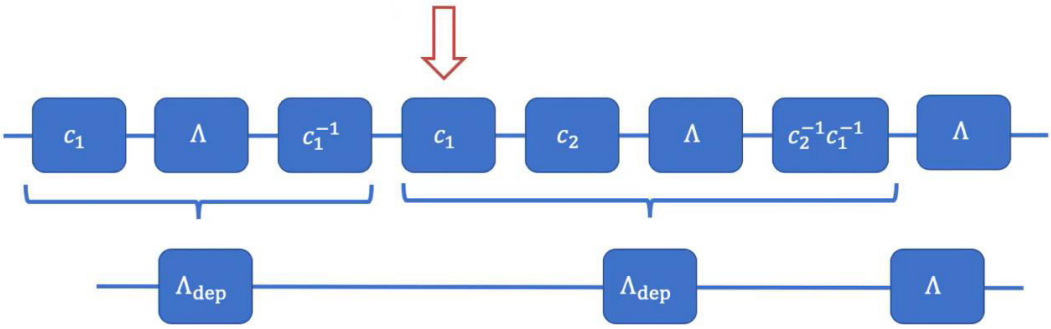
Theoretical Background of RB :

- **Ideal Clifford-gate Sequence**

- **Noisy Clifford-gate sequence for Noisy channel Λ**



$$\text{Depolarizing channel: } \Lambda_{\text{dep}}(\rho) := \frac{1}{|C_d^n|}\sum_{l=1}^{|C_d^n|} c_l^\dagger \Lambda\left(c_l \rho c_l^\dagger\right) c_l$$



- **By independence, this factorizes into product of average channels (‘*Twirl*’)**
- **Sequence of Twirled noisy channels:**

$$\Lambda \circ \Lambda_{\text{dep}}^{\circ m-1}(\rho) = p^{m-1}\Lambda(\rho) + \left(1 - p^{m-1}\right)\Lambda(\mathbb{1})/d^n$$

- **Depolarizing channel (Shur's Lemma):**

$$\Lambda(\rho) = p\rho + (1-p)/d * I$$

- **The average data looks like a product of depolarizing channels:**

$$p^m\rho + (1-p^m)/d * I$$
$$p^m(d-1)/d + 1/d$$

- **Fit the results:**

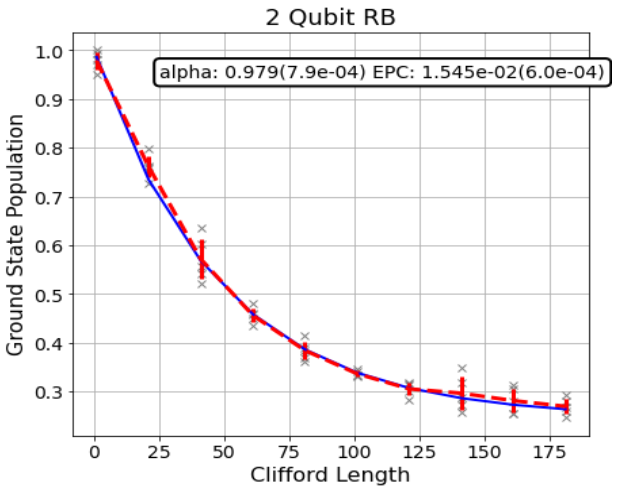
$$Ap^m + B$$

- **Calculate the average Error per Clifford (EPC):**

$$r = (1-p) - (1-p)/d$$
$$= (1-p)(d-1)/d$$

- **Calculate the 2-qubit gate fidelity:**

$$r_{cx} \approx r/n_{cx}$$
$$n_{cx} \approx 1.5$$



Theoretical Background of the Clifford group :

- The **Clifford group** C_n on n-qubits is defined as the normalizer of the **Pauli group** P_n, when neglecting the global phase U(1)= $\mathbb{C}$

$$P_n / \boldsymbol{U(1)} = \boldsymbol{F_2^{2n}}$$
$$C_n/P_n = Sp(2n)$$

- Sp(2n) denotes the group 2n×2n **symplectic** matrices over the field $\mathbf{F_2}$, which is **simple** when n>2.

- **1-qubit Clifford group (24 elements):**
  Since $C_l/P_l=Sp(2)=S_3$, we can explicitly present each elements of C_l as a product AB where:

$$A\ in\ \{I,V,W,H,HV,HW\}(V = S^\dagger H, W = HS) \quad , \quad B\ in\ P_l = \{I,X,Y,Z\}$$

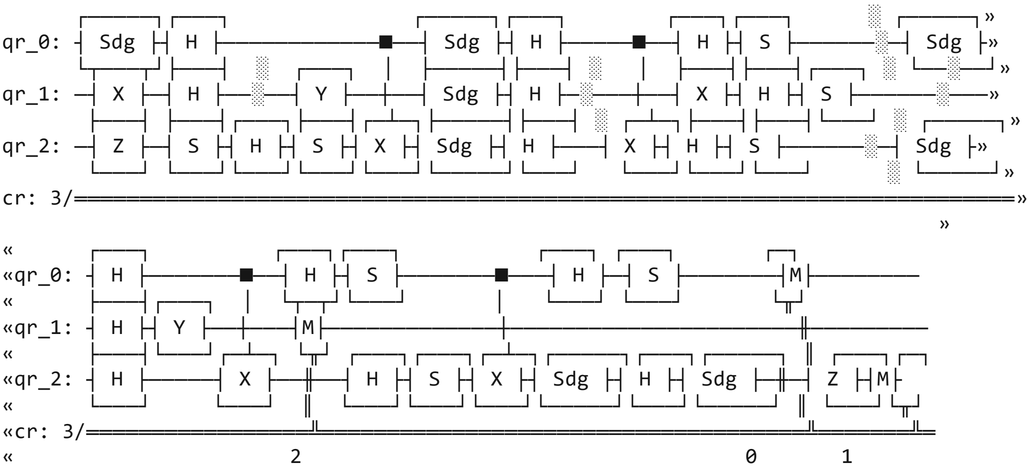
- **n-qubit Clifford group:**
  Algorithms to efficiently select random elements of C_n of size: $2^{n^2+2n}\prod_{j=1}^{n}(4^j-1)$
  Algorithms to synthesize elements of C_n (optimized for n=1,2,3)

Appendix

Python Code for Simulations

Created on Wed Jul 06 14:56:59 2022 UTC

```
#Import general libraries (needed for functions)
import numpy as np
import matplotlib.pyplot as plt
from IPython import display
#Import Qiskit classes
import qiskit
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors.standard_errors import depolarizing_error, thermal_relaxation_error

#Import the RB Functions
import qiskit.ignis.verification.randomized_benchmarking as rb

#Number of qubits
nQ = 3
#There are 3 qubits: Q0,Q1,Q2.
#Number of seeds (random sequences)
nseeds = 5
#Number of Cliffords in the sequence (start, stop, steps)
nCliffs = np.arange(1,200,20)
#2Q RB on Q0,Q2 and 1Q RB on Q1
rb_pattern = [[0,2],[1]]
#Do three times as many 1Q Cliffords
length_multiplier = [1,3]

rb_opts = {}
rb_opts['length_vector'] = nCliffs
rb_opts['nseeds'] = nseeds
rb_opts['rb_pattern'] = rb_pattern
rb_opts['length_multiplier'] = length_multiplier
rb_circs, xdata = rb.randomized_benchmarking_seq(**rb_opts)
print(rb_circs[0][0])
```



```
#Create a new circuit without the measurement
qc = qiskit.QuantumCircuit(*rb_circs[0][-1].qregs,*rb_circs[0][-1].cregs)
for i in rb_circs[0][-1][0:-nQ]:
    qc.data.append(i)

#The Unitary is an identity (with a global phase)
backend = qiskit.Aer.get_backend('unitary_simulator')
basis_gates = ['u1', 'u2', 'u3', 'cx'] # use U,CX for now
job = qiskit.execute(qc, backend=backend, basis_gates=basis_gates)
print(np.around(job.result().get_unitary(), 3))
```

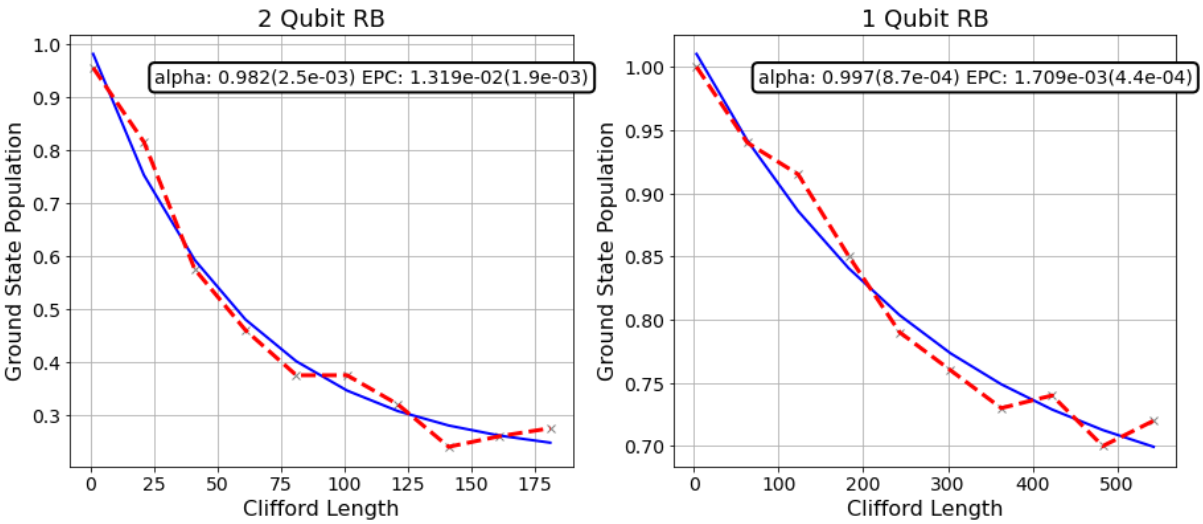
```
[[-1.+0.j -0.-0.j -0.-0.j -0.-0.j  0.+0.j  0.-0.j -0.-0.j  0.+0.j]
 [ 0.-0.j -1.+0.j -0.+0.j -0.-0.j -0.+0.j -0.+0.j -0.-0.j  0.+0.j]
 [ 0.-0.j -0.-0.j -1.+0.j -0.-0.j -0.-0.j -0.+0.j -0.+0.j -0.-0.j]
 [-0.-0.j  0.+0.j  0.-0.j -1.+0.j  0.-0.j -0.-0.j -0.+0.j  0.-0.j]
 [ 0.+0.j -0.+0.j -0.-0.j  0.+0.j -1.+0.j -0.+0.j -0.-0.j -0.-0.j]
 [ 0.-0.j -0.-0.j -0.+0.j -0.-0.j  0.-0.j -1.+0.j -0.+0.j -0.-0.j]
 [ 0.-0.j -0.+0.j -0.+0.j  0.+0.j  0.-0.j -0.-0.j -1.+0.j -0.-0.j]
 [-0.+0.j  0.+0.j  0.+0.j -0.-0.j -0.-0.j  0.+0.j  0.+0.j -1.+0.j]]
```

```
noise_model = NoiseModel()
p1Q = 0.002
p2Q = 0.01
noise_model.add_all_qubit_quantum_error(depolarizing_error(p1Q, 1), 'u2')
noise_model.add_all_qubit_quantum_error(depolarizing_error(2*p1Q, 1), 'u3')
noise_model.add_all_qubit_quantum_error(depolarizing_error(p2Q, 2), 'cx')

backend = qiskit.Aer.get_backend('qasm_simulator')
basis_gates = ['u1','u2','u3','cx'] # use U,CX for now
shots = 200
result_list = []
transpile_list = []
import time
for rb_seed,rb_circ_seed in enumerate(rb_circs):
    print('Compiling seed %d'%rb_seed)
    rb_circ_transpile = qiskit.transpile(rb_circ_seed, basis_gates=basis_gates)
    print('Simulating seed %d'%rb_seed)
    job = qiskit.execute(rb_circ_transpile, noise_model=noise_model, shots=shots, backend=backend,
backend_options={'max_parallel_experiments': 0})
    result_list.append(job.result())
    transpile_list.append(rb_circ_transpile)
print("Finished Simulating")
```

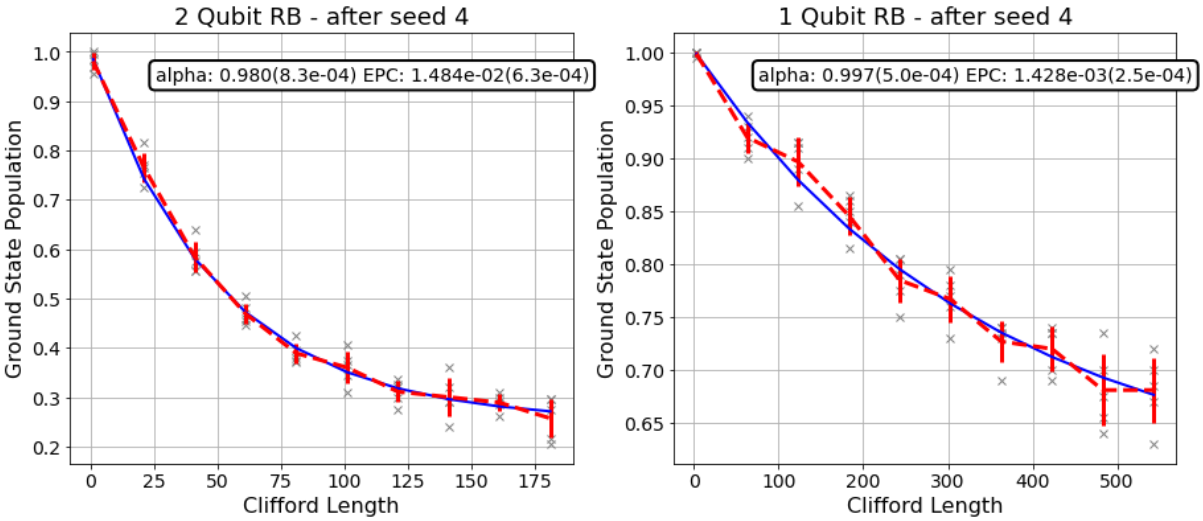
```
Compiling seed 0
Simulating seed 0
Compiling seed 1
Simulating seed 1
Compiling seed 2
Simulating seed 2
Compiling seed 3
Simulating seed 3
Compiling seed 4
Simulating seed 4
Finished Simulating
```

```
#Create an RBFitter object with 1 seed of data
rbfit = rb.fitters.RBFitter(result_list[0], xdata, rb_opts['rb_pattern'])


plt.figure(figsize=(15, 6))
```

```
for i in range(2):
    ax = plt.subplot(1, 2, i+1)
    pattern_ind = i

    # Plot the essence by calling plot_rb_data
    rbfit.plot_rb_data(pattern_ind, ax=ax, add_label=True, show_plt=False)

    # Add title and label
    ax.set_title('%d Qubit RB'%(len(rb_opts['rb_pattern'][i])), fontsize=18)

plt.show()
```



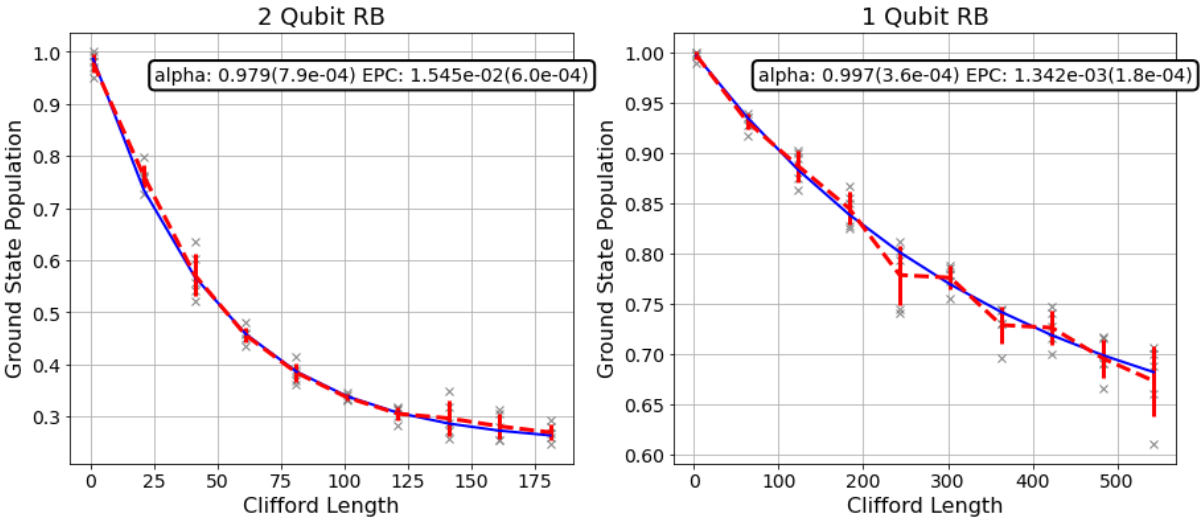

```
rbfit = rb.fitters.RBFitter(result_list[0], xdata, rb_opts['rb_pattern'])

for seed_num, data in enumerate(result_list):#range(1,len(result_list)):
    plt.figure(figsize=(15, 6))
    axis = [plt.subplot(1, 2, 1), plt.subplot(1, 2, 2)]

    # Add another seed to the data
    rbfit.add_data([data])

    for i in range(2):
        pattern_ind = i

        # Plot the essence by calling plot_rb_data
        rbfit.plot_rb_data(pattern_ind, ax=axis[i], add_label=True, show_plt=False)

        # Add title and label
        axis[i].set_title('%d Qubit RB - after seed %d'%(len(rb_opts['rb_pattern'][i]), seed_num), fontsize=18)

    # Display
    display.display(plt.gcf())

    # Clear display after each seed and close
    display.clear_output(wait=True)
    time.sleep(1.0)
    plt.close()
```




```
shots = 200
result_list = []
transpile_list = []
for rb_seed,rb_circ_seed in enumerate(rb_circs):
    print('Compiling seed %d'%rb_seed)
    rb_circ_transpile = qiskit.transpile(rb_circ_seed, basis_gates=basis_gates)
    print('Simulating seed %d'%rb_seed)
    job = qiskit.execute(rb_circ_transpile, noise_model=noise_model, shots=shots, backend=backend,
backend_options={'max_parallel_experiments': 0})
    result_list.append(job.result())
    transpile_list.append(rb_circ_transpile)
print("Finished Simulating")
```

```
Compiling seed 0
Simulating seed 0
Compiling seed 1
Simulating seed 1
Compiling seed 2
Simulating seed 2
Compiling seed 3
Simulating seed 3
Compiling seed 4
Simulating seed 4
Finished Simulating
```

```
#Add this data to the previous fit
rbfit.add_data(result_list)

#Replot
plt.figure(figsize=(15, 6))

for i in range(2):
    ax = plt.subplot(1, 2, i+1)
    pattern_ind = i

    # Plot the essence by calling plot_rb_data
    rbfit.plot_rb_data(pattern_ind, ax=ax, add_label=True, show_plt=False)

    # Add title and label
    ax.set_title('%d Qubit RB'%(len(rb_opts['rb_pattern'][i])), fontsize=18)

plt.show()
```




```
#Count the number of single and 2Q gates in the 2Q Cliffords
gates_per_cliff = rb.rb_utils.gates_per_clifford(transpile_list,xdata[0],basis_gates,rb_opts['rb_pattern'][0])
for basis_gate in basis_gates:
    print("Number of %s gates per Clifford: %f "%(basis_gate ,
                                                  np.mean([gates_per_cliff[0][basis_gate],
                                                           gates_per_cliff[2][basis_gate]])))
```

```
Number of u1 gates per Clifford: 0.142174
Number of u2 gates per Clifford: 1.643696
Number of u3 gates per Clifford: 0.166957
Number of cx gates per Clifford: 1.468913
```

```
# Error per gate from noise model
epgs_1q = {'u1': 0, 'u2': p1Q/2, 'u3': 2*p1Q/2}
epg_2q = p2Q*3/4
pred_epc = rb.rb_utils.calculate_2q_epc(
    gate_per_cliff=gates_per_cliff,
    epg_2q=epg_2q,
    qubit_pair=[0, 2],
    list_epgs_1q=[epgs_1q, epgs_1q])
```

```
# Calculate the predicted epc
print("Predicted 2Q Error per Clifford: %e"%pred_epc)
```

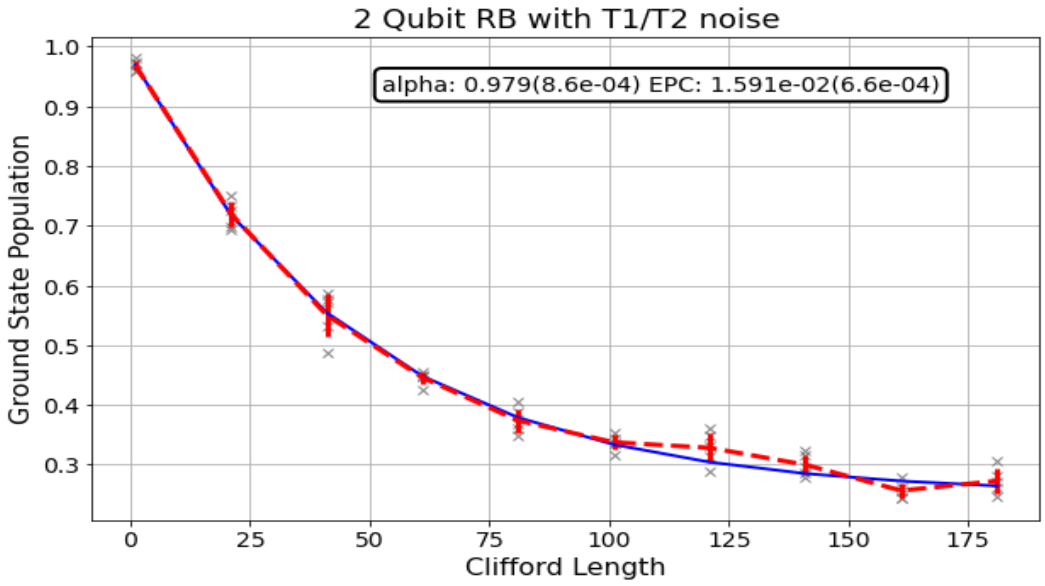
```
Predicted 2Q Error per Clifford: 1.565697e-02
```

```
rb_opts2 = rb_opts.copy()
rb_opts2['rb_pattern'] = [[0,1]]
rb_opts2['length_multiplier'] = 1
rb_circs2, xdata2 = rb.randomized_benchmarking_seq(**rb_opts2)

noise_model2 = NoiseModel()

#Add T1/T2 noise to the simulation
t1 = 100.
t2 = 80.
gate1Q = 0.1
gate2Q = 0.5
noise_model2.add_all_qubit_quantum_error(thermal_relaxation_error(t1,t2,gate1Q), 'u2')
noise_model2.add_all_qubit_quantum_error(thermal_relaxation_error(t1,t2,2*gate1Q), 'u3')
noise_model2.add_all_qubit_quantum_error(
    thermal_relaxation_error(t1,t2,gate2Q).tensor(thermal_relaxation_error(t1,t2,gate2Q)), 'cx')
```

```
backend = qiskit.Aer.get_backend('qasm_simulator')
basis_gates = ['u1','u2','u3','cx'] # use U,CX for now
shots = 500
result_list2 = []
transpile_list2 = []
for rb_seed,rb_circ_seed in enumerate(rb_circs2):
    print('Compiling seed %d'%rb_seed)
    rb_circ_transpile = qiskit.transpile(rb_circ_seed, basis_gates=basis_gates)
    print('Simulating seed %d'%rb_seed)
    job = qiskit.execute(rb_circ_transpile, noise_model=noise_model, shots=shots, backend=backend,
backend_options={'max_parallel_experiments': 0})
    result_list2.append(job.result())
    transpile_list2.append(rb_circ_transpile)
print("Finished Simulating")
```

```
Compiling seed 0
Simulating seed 0
Compiling seed 1
Simulating seed 1
Compiling seed 2
Simulating seed 2
Compiling seed 3
Simulating seed 3
Compiling seed 4
Simulating seed 4
Finished Simulating
```

```
#Create an RBFitter object
rbfit = rb.RBFitter(result_list2, xdata2, rb_opts2['rb_pattern'])

plt.figure(figsize=(10, 6))
ax = plt.gca()

# Plot the essence by calling plot_rb_data
rbfit.plot_rb_data(0, ax=ax, add_label=True, show_plt=False)

# Add title and label
ax.set_title('2 Qubit RB with T1/T2 noise', fontsize=18)

plt.show()
```



```
#Count the number of single and 2Q gates in the 2Q Cliffords
gates_per_cliff = rb.rb_utils.gates_per_clifford(transpile_list2,xdata[0],basis_gates,rb_opts2['rb_pattern'][0])
for basis_gate in basis_gates:
   print("Number of %s gates per Clifford: %f "%(basis_gate ,
                                    np.mean([gates_per_cliff[0][basis_gate],
                                          gates_per_cliff[1][basis_gate]])))
```

```
Number of u1 gates per Clifford: 0.119674
Number of u2 gates per Clifford: 1.655000
Number of u3 gates per Clifford: 0.181848
Number of cx gates per Clifford: 1.500652
```

```
# Predicted primitive gate errors from the coherence limit
u2_error = rb.rb_utils.coherence_limit(1,[t1],[t2],gate1Q)
u3_error = rb.rb_utils.coherence_limit(1,[t1],[t2],2*gate1Q)
epg_2q = rb.rb_utils.coherence_limit(2,[t1,t1],[t2,t2],gate2Q)
epgs_1q = {'u1': 0, 'u2': u2_error, 'u3': u3_error}
pred_epc = rb.rb_utils.calculate_2q_epc(
   gate_per_cliff=gates_per_cliff,
   epg_2q=epg_2q,
   qubit_pair=[0, 1],
   list_epgs_1q=[epgs_1q, epgs_1q])

# Calculate the predicted epc
print("Predicted 2Q Error per Clifford: %e"%pred_epc)
```

```
Predicted 2Q Error per Clifford: 1.320763e-02
```

```
import qiskit.tools.jupyter
%qiskit_version_table
%qiskit_copyright
```

Qiskit Software	Version
qiskit-terra	0.20.2
qiskit-aer	0.10.4
qiskit-ignis	0.7.1
qiskit-ibmq-provider	0.19.1
qiskit	0.36.2
qiskit-nature	0.4.1
qiskit-finance	0.3.3
qiskit-optimization	0.4.0
qiskit-machine-learning	0.4.0

System information

```
Python version      3.8.13
Python compiler GCC 10.3.0
Python build default, Mar 25 2022 06:04:10
   OS               Linux
CPUs8Memory (Gb)31.211315155029297
Wed Jul 06 14:56:59 2022 UTC
```

References

1.Scott Aaaronsonand Daniel Gottesman, *Improved Simulation of Stabilizer Circuits*, Phys. Rev. A 70(052328), 2004. https://arxiv.org/abs/quant-ph/0406196
2.Mahnaz Jaafarzadeh Randomized Benchmarking For Qudit Clifford Gates el al 2020 New J. Phys.. 22 063014 https://iopscience.iop.org/article/10.1088/1367-2630/ab8ab1
3.Timothy J. Proctor, Arnaud Carignan-Dugas, Kenneth Rudinger, Erik Nielsen, Robin Blume-Kohout, and Kevin Young, *Direct randomized benchmarking for multi-qubit devices*, Phys. Rev. Lett. 12(030503) 2019,https://arxiv.org/abs/1807.07975
4.Shelly Garionand Andrew W. Cross, *On the Structure of the Non-Clifford CNOT-Dihedral Group*, *in preparation.*
5. https://qiskit.org/documentation/stable/0.24/tutorials/noise/4_randomized_benchmarking.html