Efficient Simulation of Clifford Circuits

Utkarsh Azad

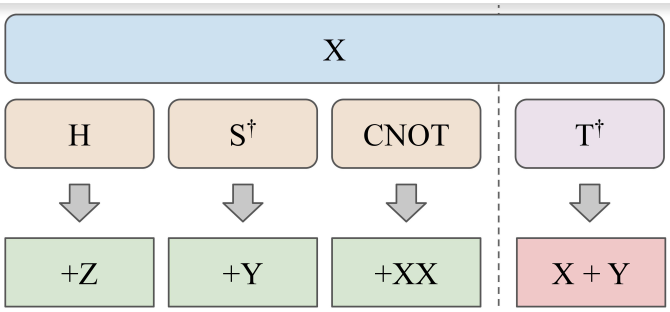
Published: April 11, 2024. Last updated: November 13, 2024.

Classical simulation of quantum circuits doesn't always require an exponential amount of computational resources. They can be performed efficiently if there exists a classical description that enables evolving the quantum state by unitary operations and performing measurements in a polynomial number of steps [1], [2]. In this tutorial, we take a deep dive into learning about Clifford circuits, which are known to be efficiently simulable by classical computers and play an essential role in the practical implementation of quantum computation. We will learn how to use PennyLane to simulate these circuits scaling up to more than thousands of qubits. We also look at the ability to decompose quantum circuits into a set of universal quantum gates comprising Clifford gates.

Clifford Group and Clifford Gates

In classical computation, one can define a universal set of logic gate operations such as `{AND, NOT, OR}` that can be used to perform any boolean function. A similar analogue in quantum computation is to have a set of quantum gates that can approximate any unitary transformation up to the desired accuracy. One such universal quantum gate set is the $\mathrm{Clifford} + \mathrm{T}$ set, `{H, S, CNOT, T}`, where the gates `H`, `S`, and `CNOT` are the generators of the *Clifford group*. The elements of this group are called *Clifford gates*, which transform *Pauli* words to *Pauli* words under conjugation. This means an n-qubit unitary C belongs to the Clifford group if the conjugates $CPC^\dagger$ are also Pauli words for all n-qubit Pauli words P. We can see this ourselves by conjugating the Pauli X operation with the elements of the universal set defined above:

PENNYLANE



Clifford gates can be obtained by combining the generators of the Clifford group along with their inverses and include the following quantum operations, which are all supported in PennyLane:

1. Single-qubit Pauli gates: **`I`**, **`X`**, **`Y`**, and **`Z`**.
2. Other single-qubit gates: **`S`** and **`Hadamard`**.
3. The two-qubit `controlled` Pauli gates: **`CNOT`**, **`CY`**, and **`CZ`**.
4. Other two-qubit gates: **`SWAP`** and **`ISWAP`**.
5. Adjoints of the above gate operations via **`adjoint()`**.

Each of these gates can be uniquely described by how they transform the Pauli words. For example, `Hadamard` conjugates X to Z and Z to X. Similarly, `ISWAP` acting on a subspace of qubits *i* and *j* conjugates X_i to Z_iY_j and Z_i to Z_j. These transformations can be presented in tabulated forms called *Clifford tableaus*, as shown below:

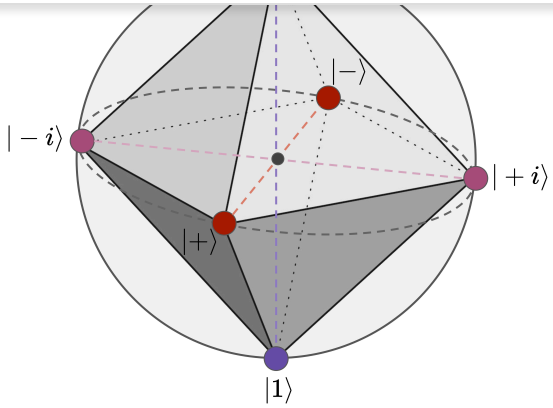
Clifford Tableau: Hadamard	
Pauli	Conjugate
X_i	$+Z_i$
Z_i	$+X_i$

Clifford Tableau: ISWAP	
Pauli	Conjugate
X_i / X_j	$+Z_iY_j$ / $+Y_iZ_j$
Z_i / Z_j	$+Z_j$ / $+Z_i$

Clifford circuits and Stabilizer Tableaus

The quantum circuits that consist only of Clifford gates are called *Clifford circuits* or, more generally, Clifford group circuits. Moreover, the Clifford circuits that also have single-qubit measurements are known as *stabilizer circuits*. The states such circuits can evolve to are known as the *stabilizer states*. For example, the following figure shows the single-qubit stabilizer states:

PENNYLANE



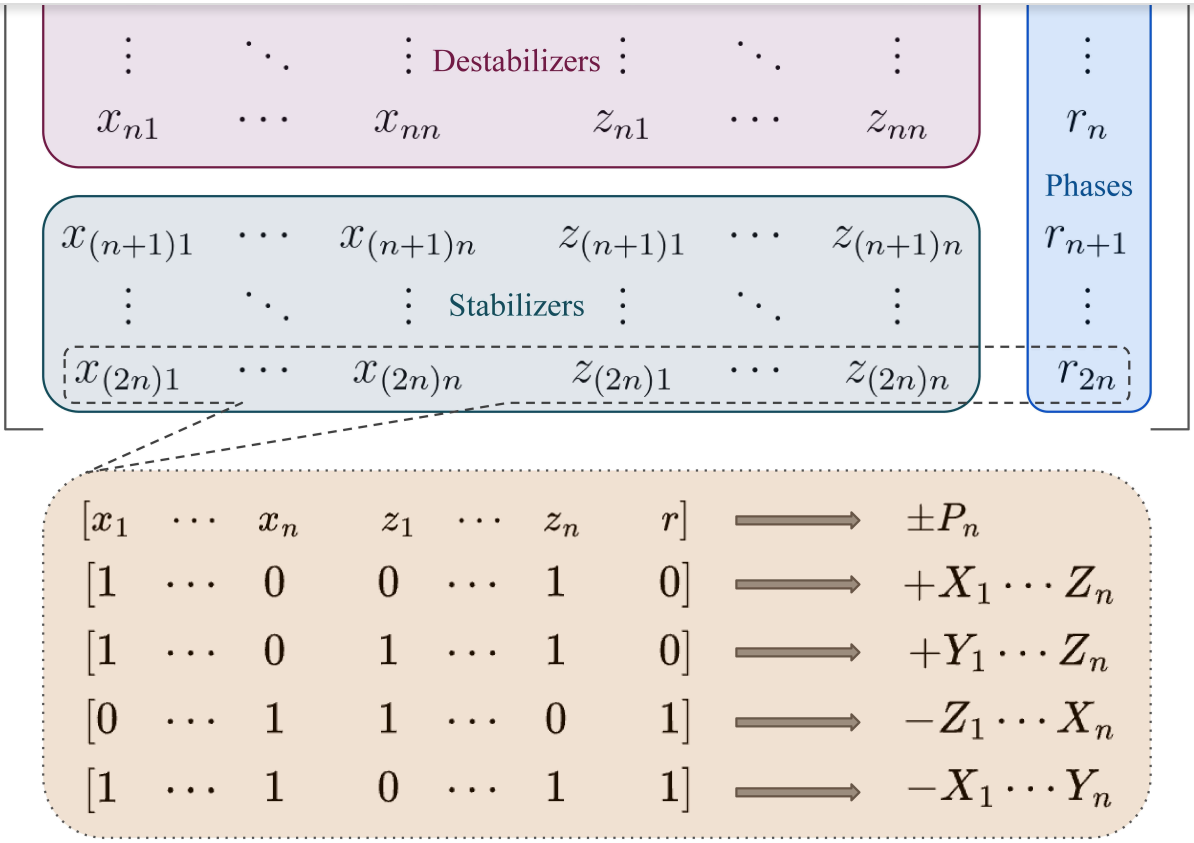
The octahedron in the Bloch sphere defines the states accessible via single-qubit Clifford gates.

These types of circuits represent extremely important classes of quantum circuits in the context of quantum error correction and measurement-based quantum computation [3]. More importantly, they can be efficiently simulated classically, according to the *Gottesman-Knill* theorem, which states that any n-qubit Clifford circuit with m Clifford gates can be simulated in time $poly(m, n)$ on a probabilistic classical computer.

There are several ways for representing n-qubit stabilizer states $|\psi\rangle$ and tracking their evolution with a $poly(n)$ number of bits. The *CHP* (CNOT-Hadamard-Phase) formalism, also called the *phase-sensitive* formalism, is one of these methods, where one efficiently describes the state using a *Stabilizer tableau* structure based on its `stabilizer` set $\mathcal{S}$. The *stabilizers,* represented by the elements `s` in $\mathcal{S}$, are n-qubit Pauli words that have the state $|\psi\rangle$ as their $+1$ eigenstate, i.e., $s|\psi\rangle = |\psi\rangle, \forall s \in \mathcal{S}$. These are often viewed as virtual `Z` operators, while their conjugates, termed *destabilizers* (`d`), correspond to virtual `X` operators, forming a similar set referred to as `destabilizer` set $\mathcal{D}$.

The stabilizer tableau for an n-qubit state is made of binary variables representing the Pauli words for the `generators` of stabilizer $\mathcal{S}$ and destabilizer $\mathcal{D}$.` These are generally arranged as the following tabulated structure [4]:

PENNYLANE



Here, the first and last n rows represent the generators d_i and s_i as check vectors, respectively, and they generate the entire Pauli group $\mathcal{P}_n$ together. The last column contains the binary variable *r* corresponding to the phase of each generator and gives the sign ($\pm$) for the Pauli word that represents them.

For evolving the state, i.e., replicating the application of the Clifford gates on the state, we update each generator and the corresponding phase according to the Clifford tableau described above [5]. In the Simulating Stabilizer Tableau section, we will expand on this in greater detail.

Clifford Simulations in PennyLane

PennyLane has a `default.clifford` device that enables efficient simulation of large-scale Clifford circuits. The device uses the stim simulator [6] as an underlying backend and can be used to run Clifford circuits in the same way we run any other regular circuits in the PennyLane ecosystem. Let's look at an example that constructs a Clifford circuit and performs several measurements.

PENNYLANE

```
dev = qml.device("default.clifford", wires=2, tableau=True)

@qml.qnode(dev)
def circuit(return_state=True):
    qml.X(wires=[0])
    qml.CNOT(wires=[0, 1])
    qml.Hadamard(wires=[0])
    qml.Hadamard(wires=[1])
    return [
        qml.expval(op=qml.X(0) @ qml.X(1)),
        qml.var(op=qml.Z(0) @ qml.Z(1)),
        qml.probs(),
    ] + ([qml.state()] if return_state else [])


expval, var, probs, state = circuit(return_state=True)
print(expval, var)
```

Out:

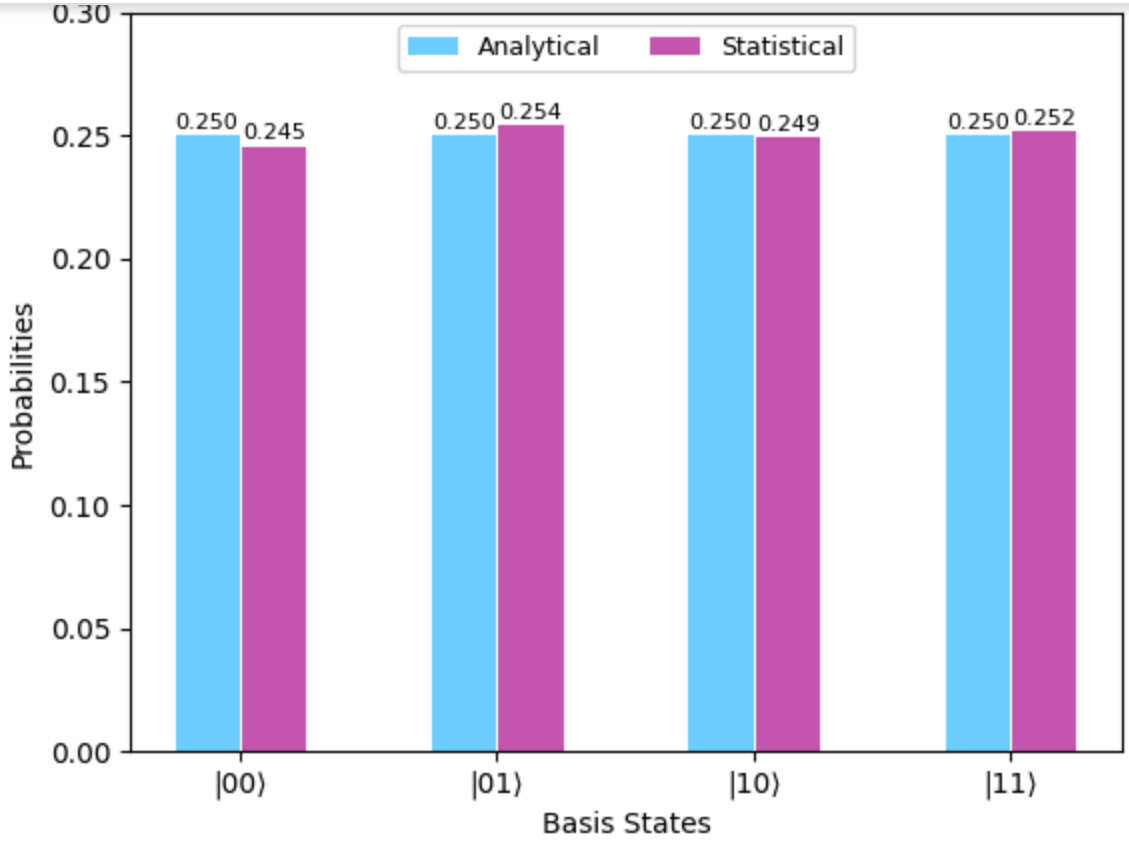
```
1.0 1.0
```

As observed, the full range of PennyLane measurements like **`expval()`** and **`probs()`** can be performed analytically with this device. Additionally, we support all the sample-based measurements on this device, similar to `default.qubit`. For instance, we can simulate the circuit with $10,000$ shots and compare the results obtained from sampling with the analytic case:

PENNYLANE

```
import matplotlib.pyplot as plt

# Get the results with 10000 shots and assert them
shot_result = circuit(return_state=False, shots=10000)
shot_exp, shot_var, shot_probs = shot_result
assert qml.math.allclose([shot_exp, shot_var], [expval, var], atol=1e-3)

# Define computational basis states
basis_states = ["|00⟩", "|01⟩", "|10⟩", "|11⟩"]

# Plot the probabilities
bar_width, bar_space = 0.25, 0.01
colors = ["#70CEFF", "#C756B2"]
labels = ["Analytical", "Statistical"]
for idx, prob in enumerate([probs, shot_probs]):
    bars = plt.bar(
        np.arange(4) + idx * (bar_width + bar_space), prob,
        width=bar_width, label=labels[idx], color=colors[idx],
    )
    plt.bar_label(bars, padding=1, fmt="%.3f", fontsize=8)

# Add labels and show
plt.title("Comparing Probabilities from Circuits", fontsize=11)
plt.xlabel("Basis States")
plt.ylabel("Probabilities")
plt.xticks(np.arange(4) + bar_width / 2, basis_states)
plt.ylim(0.0, 0.30)
plt.legend(loc="upper center", ncols=2, fontsize=9)
plt.show()
```

PENNYLANE



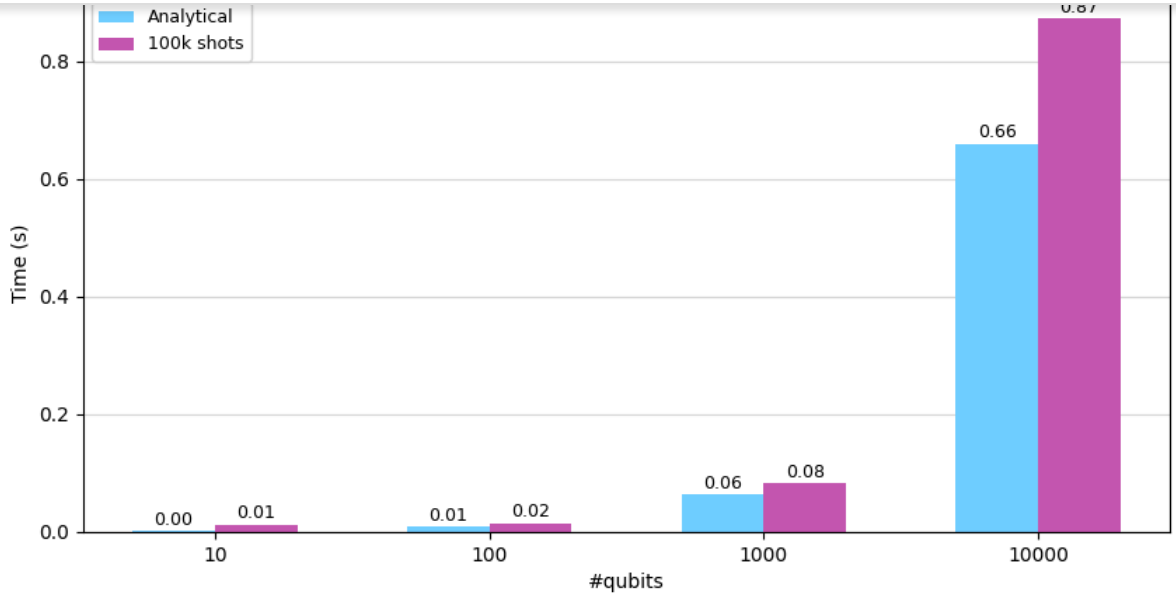
Benchmarking

Now that we've had a slight taste of what `default.clifford` can do, let's push the limits to benchmark its capabilities. To achieve this, we'll examine a set of experiments with the n-qubits Greenberger-Horne-Zeilinger state (GHZ state) preparation circuit:

```
dev = qml.device("default.clifford")

@qml.qnode(dev)
def GHZStatePrep(num_wires):
    """Prepares the GHZ State"""
    qml.Hadamard(wires=[0])
    for wire in range(num_wires):
        qml.CNOT(wires=[wire, wire + 1])
    return qml.expval(qml.Z(0) @ qml.Z(num_wires - 1))
```

PENNYLANE

```
dev = qml.device("default.clifford")

num_shots = [None, 100000]
num_wires = [10, 100, 1000, 10000]

shots_times = np.zeros((len(num_shots), len(num_wires)))

# Iterate over different number of shots and wires
for ind, num_shot in enumerate(num_shots):
    for idx, num_wire in enumerate(num_wires):
        shots_times[ind][idx] = timeit(
            "GHZStatePrep(num_wire, shots=num_shot)", number=5, globals=globals()
        ) / 5 # average over 5 trials

# Figure set up
fig = plt.figure(figsize=(10, 5))

# Plot the data
bar_width, bar_space = 0.3, 0.01
colors = ["#70CEFF", "#C756B2"]
labels = ["Analytical", "100k shots"]
for idx, num_shot in enumerate(num_shots):
    bars = plt.bar(
        np.arange(len(num_wires)) + idx * bar_width, shots_times[idx],
        width=bar_width, label=labels[idx], color=colors[idx],
    )
    plt.bar_label(bars, padding=1, fmt="%.2f", fontsize=9)

# Add labels and titles
plt.xlabel("#qubits")
plt.ylabel("Time (s)")
plt.gca().set_axisbelow(True)
plt.grid(axis="y", alpha=0.5)
plt.xticks(np.arange(len(num_wires)) + bar_width / 2, num_wires)
plt.title("Execution Times with varying shots")
plt.legend(fontsize=9)
plt.show()
```



These results clearly demonstrate that large-scale analytic and sampling simulations can be performed using `default.clifford`. Remarkably, the computation time remains consistent, particularly when the number of qubits scales up, making it evident that this device significantly outperforms state vector-based devices like `default.qubit` or `lightning.qubit` for simulating stabilizer circuits.

Simulating Stabilizer Tableau

Looking at the benchmarks, one may want to delve into understanding what makes the underlying stabilizer tableau formalism performant. To do this, we need to access the state of the device as a stabilizer tableau using the **`state()`** function. This can be done if the device is initialized with the default `tableau=True` keyword argument. For example, the tableau for the above circuit is:

```
print(state)
```

Out:

```
[[0 0 1 1 0]
 [0 0 0 1 0]
 [1 0 0 0 1]
 [1 1 0 0 0]]
```

PENNYLANE

them. This approach offers a more human-readable visualization of tableaus and the following methods assist us in achieving it. Let's generate the stabilizers and destabilizers of the state obtained above.

```
from pennylane.pauli import binary_to_pauli, pauli_word_to_string

def tableau_to_pauli_group(tableau):
    """Get stabilizers, destabilizers and phase from a Tableau"""
    num_qubits = tableau.shape[0] // 2
    stab_mat, destab_mat = tableau[num_qubits:, :-1], tableau[:num_qubits, :-1]
    stabilizers = [binary_to_pauli(stab) for stab in stab_mat]
    destabilizers = [binary_to_pauli(destab) for destab in destab_mat]
    phases = tableau[:, -1].reshape(-1, num_qubits).T
    return stabilizers, destabilizers, phases

def tableau_to_pauli_rep(tableau):
    """Get a string representation for stabilizers and destabilizers from a Table
    wire_map = {idx: idx for idx in range(tableau.shape[0] // 2)}
    stabilizers, destabilizers, phases = tableau_to_pauli_group(tableau)
    stab_rep, destab_rep = [], []
    for phase, stabilizer, destabilizer in zip(phases, stabilizers, destabilizers
        p_rep = ["+" if not p else "-" for p in phase]
        stab_rep.append(p_rep[1] + pauli_word_to_string(stabilizer, wire_map))
        destab_rep.append(p_rep[0] + pauli_word_to_string(destabilizer, wire_map)
    return {"Stabilizers": stab_rep, "Destabilizers": destab_rep}

tableau_to_pauli_rep(state)
```

Out:

```
{'Stabilizers': ['-XI', '+XX'], 'Destabilizers': ['+ZZ', '+IZ']}
```

As previously suggested, the evolution of the stabilizer tableau after the application of each Clifford gate operation can be understood by learning how the generator set is transformed based

PENNYLANE

```
def clifford_tableau(op):
    """Prints a Clifford Tableau representation for a given operation."""
    # Print the op and set up Pauli operators to be conjugated
    print(f"Tableau: {op.name}({', '.join(map(str, op.wires))})")
    pauli_ops = [pauli(wire) for wire in op.wires for pauli in [qml.X, qml.Z]]
    # obtain conjugation of Pauli op and decompose it in Pauli basis
    for pauli in pauli_ops:
        conjugate = qml.prod(qml.adjoint(op), pauli, op).simplify()
        decompose = qml.pauli_decompose(conjugate.matrix(), wire_order=op.wires)
        decompose_coeffs, decompose_ops = decompose.terms()
        phase = "+" if list(decompose_coeffs)[0] >= 0 else "-"
        print(pauli, "-->", phase, list(decompose_ops)[0])

clifford_tableau(qml.X(0))
```

Out:

```
Tableau: PauliX(0)
X(0) --> + X(0)
Z(0) --> - Z(0)
```

We now have the two key components for studying the evolution of the stabilizer tableau of the described circuit - (i) the *generator* set representation to describe the stabilizer state, and (ii) *tableau* representation for the Clifford gate that is applied to it. However, in addition to these we would also need a method to access the circuit's state before and after the application of gate operation. This is achieved by inserting the **`snapshots()`** in the circuit using the following transform.

PENNYLANE

```
def state_at_each_step(tape):
    """Transforms a circuit to access state after every operation"""
    # This builds list with a qml.Snapshot operation before every tape operation
    operations = []
    for op in tape.operations:
        operations.append(qml.Snapshot())
        operations.append(op)
    operations.append(qml.Snapshot()) # add a final qml.Snapshot operation at ena
    new_tape = type(tape)(operations, tape.measurements, shots=tape.shots)
    postprocessing = lambda results: results[0] # func for processing results
    return [new_tape], postprocessing

snapshots = qml.snapshots(state_at_each_step(circuit))()
```

We can now access the tableau state via the `snapshots` dictionary, where the integer keys represent each step. The step `0` corresponds to the initial all zero $|00\rangle$ state, which is stabilized by the Pauli operators Z_0 and Z_1. Evolving it by a `qml.X(0)` would correspond to transforming its stabilizer generators from $+Z_0$ to $-Z_0$, while keeping the destabilizer generators the same.

```
print("Intial State: ", tableau_to_pauli_rep(snapshots[0]))
print("Applying X(0): ", tableau_to_pauli_rep(snapshots[1]))
```

Out:

```
Intial State:  {'Stabilizers': ['+ZI', '+IZ'], 'Destabilizers': ['+XI', '+
Applying X(0):  {'Stabilizers': ['-ZI', '+IZ'], 'Destabilizers': ['+XI', '
```

The process worked as anticipated! So, to track and compute the evolved state, one only needs to know the transformation rules for each gate operation described by their tableau. This makes the tableau formalism much more efficient than the state vector formalism, where a more computationally expensive matrix-vector multiplication has to be performed at each step. Let's examine the remaining operations to confirm this.

PENNYLANE

```
circuit_ops = tape.operations
print("Circ. Ops: ", circuit_ops)

for step in range(1, len(circuit_ops)):
    print("--" * 7 + f" Step {step} - {circuit_ops[step]} " + "--" * 7)
    clifford_tableau(circuit_ops[step])
    print(f"Before - {tableau_to_pauli_rep(snapshots[step])}")
    print(f"After  - {tableau_to_pauli_rep(snapshots[step+1])}\n")
```

Out:

```
Circ. Ops:  [X(0), CNOT(wires=[0, 1]), H(0), H(1)]
-------------- Step 1 - CNOT(wires=[0, 1]) --------------
Tableau: CNOT(0, 1)
X(0) -> + X(0) @ X(1)
Z(0) -> + Z(0) @ I(1)
X(1) -> + I(0) @ X(1)
Z(1) -> + Z(0) @ Z(1)
Before - {'Stabilizers': ['-ZI', '+IZ'], 'Destabilizers': ['+XI', '+IX']
After  - {'Stabilizers': ['-ZI', '+ZZ'], 'Destabilizers': ['+XX', '+IX']

-------------- Step 2 - H(0) --------------
Tableau: Hadamard(0)
X(0) -> + Z(0)
Z(0) -> + X(0)
Before - {'Stabilizers': ['-ZI', '+ZZ'], 'Destabilizers': ['+XX', '+IX']
After  - {'Stabilizers': ['-XI', '+XZ'], 'Destabilizers': ['+ZX', '+IX']
```

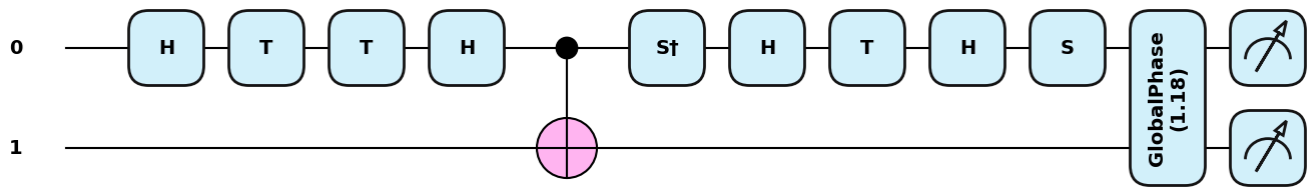
Clifford + T Decomposition

Finally, you might wonder if there's a programmatic way to determine whether a given circuit is a Clifford or a stabilizer circuit, or which gates in the circuit are non-Clifford operations. While the `default.clifford` device internally attempts this by decomposing each gate operation into the Clifford basis, one can also do this independently on their own. In PennyLane, any quantum circuit can be decomposed in a universal basis using the **`clifford_t_decomposition()`**. This transform,

PENNYLANE

described in Dawson and Nielsen (2005). Let's see this in action for the following two-qubit parameterized circuit:

```
dev = qml.device("default.qubit")
@qml.qnode(dev)
def original_circuit(x, y):
    qml.RX(x, 0)
    qml.CNOT([0, 1])
    qml.RY(y, 0)
    return qml.probs()

x, y = np.pi / 2, np.pi / 4
unrolled_circuit = qml.transforms.clifford_t_decomposition(original_circuit)

qml.draw_mpl(unrolled_circuit, decimals=2, style="pennylane")(x, y)
plt.show()
```



In the *unrolled* quantum circuit, we can see that the non-Clifford rotation gates **RX** and **RY** at the either side of **CNOT** has been replaced by the sequence of single-qubit Clifford and T gates, which depend on their parameter values. In order to ensure that the performed decomposition is correct, we can compare the measurement results of the unrolled and original circuits.

```
original_probs, unrolled_probs = original_circuit(x, y), unrolled_circuit(x, y)
assert qml.math.allclose(original_probs, unrolled_probs, atol=1e-3)
```

Ultimately, one can use this decomposition to perform some basic resource analysis for fault-tolerant quantum computation, such as calculating the number of non-Clifford T gate operations

PENNYLANE

circuit. This is due to their influence on the fault-tolerant thresholds for error correction codes, as outlined in the Eastin-Knill theorem.

```
with qml.Tracker(dev) as tracker:
    unrolled_circuit(x, y)

resources_lst = tracker.history["resources"]
print(resources_lst[0])
```

Out:

```
num_wires: 2
num_gates: 11
depth: 11
shots: Shots(total=None)
gate_types:
{'Hadamard': 4, 'T': 3, 'CNOT': 1, 'Adjoint(S)': 1, 'S': 1, 'GlobalPhase':
gate_sizes:
{1: 9, 2: 1, 0: 1}
```

Conclusion

Stabilizer circuits are an important class of quantum circuits that are used in quantum error correction and benchmarking the performance of quantum hardware. The `default.clifford` device in PennyLane enables efficient classical simulations of large-scale Clifford circuits and their use for these purposes. The device allows one to obtain the Tableau form of the quantum state and supports a wide range of essential analytical and statistical measurements such as expectation values, samples, entropy-based results and even classical shadow-based results. Additionally, it supports finite-shot execution with noise channels that add single or multi-qubit Pauli noise, such as depolarization and flip errors. PennyLane also provides a functional way to decompose and compile a circuit into a universal basis, which can ultimately enable the simulation of near-Clifford circuits.

[1] D. Maslov, S. Bravyi, F. Tripier, A. Maksymov, and J. Latone “Fast classical simulation of Harvard/QuEra IQP circuits” arXiv:2402.03211, 2024.

[2] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, and *et al.* “Classical Simulation of Quantum Supremacy Circuits” arXiv:2005.06787, 2020.

[3] H. J. Briegel, D. E. Browne, W. Dür, R. Raussendorf, and M. V. den Nest “Measurement-based quantum computation” arXiv:0910.1116, 2009.

[4] S. Bravyi, D. Browne, P. Calpin, E. Campbell, D. Gosset, and M. Howard “Simulation of quantum circuits by low-rank stabilizer decompositions” Quantum 3, 181, 2019.

[5] S. Aaronson and D. Gottesman “Improved simulation of stabilizer circuits” Phys. Rev. A 70, 052328, 2004.

[6] C. Gidney “Stim: a fast stabilizer circuit simulator” Quantum 5, 497, 2021.

About the author

Utkarsh Azad

Fractals, computing and poetry.

Total running time of the script: (0 minutes 9.591 seconds)

Ask a question on the forum

Share demo

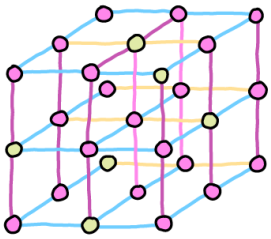
0 Comments - *powered by utteranc.es*

Write Preview

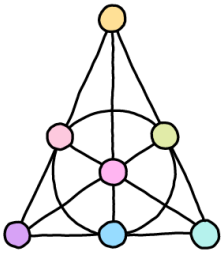
Sign in to comment

PENNYLANE

Related Demos



Measurement-based quantum computation



Unitary designs

PennyLane is an open-source software framework for quantum machine learning, quantum chemistry, and quantum computing, with the ability to run on all hardware. Built with ❤️ by Xanadu.

For researchers

Research
Features
Demos
Compilation

For learners

Learn
Codebook
Teach
Videos

For developers

Features
Documentation
API
GitHub