**Special Topics on Quantum Design Automation Final Project**

*Testing for Quantum Circuits*

**Introduction:**

Quantum testing is the procedure to decide whether the quantum hardware is functionally correct or not. The procedure includes *fault modeling*, *fault simulation,* and *test generation.* This project provides students the fundamental idea of *quantum Automatic Test Generation (qATG)* and the method of testing quantum hardware using qATG [1]. To begin with, you need to create fault models that accurately characterize the fault behavior of quantum gates. Subsequently, it is necessary to conduct fault simulation utilizing these fault models. Using both the fault-free and faulty output distributions, you can determine the *test repetitions,* given test escape and overkill criteria. Third, you should be able to judge whether the quantum hardware *passes* or *fails* your test based on a given output distribution. Finally, you are requested to do testing on four *simulated quantum backends* and three distinct fault models. You should determine which quantum backend fails or passes the test.

**Environment Setup:**

1. Install Python 3.10
2. Install qATG **(ver. 0.8.2 or ver. 0.7.0 or ver. 0.6.1)**

There are several ways to install the qATG package. The first method is to install the package from local file :

1. Download qATG from GitHub

**(Notice: Please download version 0.6.1 or version 0.7.0 or version 0.8.2)**

<https://github.com/NTU-LaDS-II/qATG>

1. Install qATG using the following comment:  
   pip install /(*file\_path\_to\_qATG*)/qATG

(For student using first method, if you need to modify the source code of qATG, please add the “**-e**” flag.)

\* NOTE: If you are using Colab, you can upload the qATG to your Google Drive and then install the qATG using the following command:  
from google.colab import drive  
drive.mount('/content/drive')  
!pip install “/content/drive/MyDrive/(*file\_path\_to\_qATG)*/qATG”

The second way is to simply install it from the GitHub repository:

pip install qatg //or qatg==**0.7.0** //or qatg==**0.6.1**

**Problem 1: Generate test configuration**

A test configuration refers to a quantum circuit that is designed to identify the occurrence of a particular fault in quantum hardware. You are requested to establish new *fault model classes* based on the provided description of fault behavior. To adhere to proper naming conventions, you are required to name your classes as *myFault\_(index)* and have them inherited from the *QATGFault* class. Three fault model classes should be established:

1. *myFault\_1* indicates the Z-direction drifting fault of the **SX gate.**
2. *myFault\_2* indicates unwanted y rotation when applying the **rotation Z gate.**
3. *myFault\_3* is the imperfect controllability of the **CNOT gate**.

Once the fault models have been defined, you are then tasked with generating test configurations for these faults using the QATG program. Importantly, ensure that the *minRequiredStateFidelity* parameter is set to a precise value of 0.1 during the test configuration generation process.

|  |  |  |
| --- | --- | --- |
| Index | Fault free gate | Faulty gate |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |

**Problem 2.** **Fault simulation**

Fault simulation is the process of simulating the behavior of a quantum circuit in the event of a particular fault. Your task is to construct a function named *fault\_simulation* that takes three inputs: a *QATGFault* object named *f*, a *QuantumCircuit* object named *qc*, and an integer named *s*. The function should return the result of executing the quantum circuit *qc* by *s* times in the presence of fault *f*. The interface of the function should be defined as follows:

def fault\_simulation(fault\_model, qc, shot):

### Fault simulation ###

return *result\_counts*

For example, consider the three fault models mentioned earlier (*myFault\_1*, *myFault\_2*, and *myFault\_3*), along with the two quantum circuits provided in *benchmarks/qc1.qasm* and *benchmarks/qc2.qasm*. Your fault simulation function should produce an output distribution that closely resembles the distribution in the following table when the 'shots' parameter is set to 100,000.

|  |  |  |
| --- | --- | --- |
| Fault Index | *qc1.qasm* result | *qc2.qasm* result |
| Fault-free | {'10': 72941, '01': 27059} | {'01': 5913, '11': 18076, '00': 38797, '10': 37214} |
| 1 | {'11': 165, '00': 411, '10': 72558, '01': 26866} | {'01': 8805, '00': 41107, '10': 36784, '11': 13304} |
| 2 | {'11': 762, '01': 25476, '00': 3160, '10': 70602} | {'00': 24819, '10': 49844, '11': 13537, '01': 11800} |
| 3 | {'11': 2087, '01': 28597, '00': 1102, '10': 68214} | {'01': 3053, '10': 49390, '00': 22959, '11': 24598} |

**Problem 3:** **Fault detection**

Your task is to design a function named *fault\_detection*, which takes the following inputs: a fault model (*fault\_model*), a quantum circuit (*qc*), an output distribution (*distribution*), a maximum test escape rate (*maximum\_test\_escape*), and a maximum overkill rate (*maximum \_overkill*). The function should return ‘*True’* if the fault is detected, and ‘*False*’ otherwise. Assumptions can be made that the fault is the only error source in the quantum circuit, with no additional noise.

Here is a function interface for your implementation:

def fault\_detection (fault\_model, qc, distribution, maximum\_test\_escape, maximum\_overkill):

# Implement fault detection logic

# Return True if the fault is detected, False otherwise

return detection\_result

While one method for fault detection is applying the revised chi-square test [1], you are welcome to explore alternative approaches, as long as they meet the specified target test escape and target overkill. Your report should clearly detail the rationale behind your chosen method, outlining how it ensures reliable fault detection.

**Problem 4: Testing**

Your objective is to execute the entire sequence of testing procedures. Specifically, you are tasked with the following steps:

* 1. Generate Test Configuration:

Create a test configuration for three specified target fault models. Ensure that the fault models are named and inherited appropriately, following the instructions from Problem 1.

4.2 Conduct Tests on Quantum Backend Simulators:

Utilize four quantum backend simulators provided in the directory *CUT(Circuit Under Tests)*  to execute tests. Record the results in the table, indicating whether each test is passed or failed. The results should be determined by the fault detection method implemented in Problem 3.

4.3 Report:

You should show your result in a table as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Quantum Backend Simulator | Test Configuration 1 | Test Configuration 2 | Test Configuration 3 |
| Backend 1 | Passed/Failed | Passed/Failed | Passed/Failed |
| Backend 2 | Passed/Failed | Passed/Failed | Passed/Failed |
| Backend 3 | Passed/Failed | Passed/Failed | Passed/Failed |
| Backend 4 | Passed/Failed | Passed/Failed | Passed/Failed |

Submit a report summarizing the overall testing process, highlighting any insights gained, challenges encountered, and the efficacy of the chosen fault detection method. Discuss any variations in test outcomes across different backend simulators. Ensure the implementation is clear and well-documented, for a comprehensive evaluation of the testing process.

Here is an example of importing the provided backend

|  |
| --- |
| from subprocess import run, PIPE  def get\_OPD(qasm\_file\_name, shots):  out = run(['python', ' CUT/backend\_1.pyc', qasm\_file\_name, str(shots)], stdout=PIPE)  return eval(out.stdout)  # example of executing quantum circuit in 'Test1.qasm' for 1024 shots  get\_OPD('Test1.qasm', 1024) |

**Requirements:**

You are required to submit the following files:

Problem 1:

*Fault\_model.py* contains three fault models above.

*Sx\_test.qasm* contains test configuration for fault 1 in qasm form.

*Rz\_test.qasm* contains test configuration for fault 2 in qasm form.

*Cnot\_test.qasm* contains test configuration for fault 3 in qasm form.

Problem 2:

*fault\_simulation.py* contains a function that performs fault simulation.

Problem 3:

*fault\_detection.py* contains a function determining whether a distribution passes or fails the test.

Problem 4:

*Test1.qasm* contains test configuration for fault 1 in qasm form.

*Test2.qasm* contains test configuration for fault 2 in qasm form.

*Test3.qasm* contains test configuration for fault 3 in qasm form.

*report.pdf* includes your results, findings, and insights gained during testing. Provide details on the implemented fault detection methods, challenges faced, and the overall effectiveness of your approach.

**References**

[1] Wu, Chen-Hung, et al. "qatg: Automatic test generation for quantum circuits." *2020 IEEE International Test Conference (ITC)*. IEEE, 2020.