Class Bit:

This class stands for the classical bit; once a qubit have been measured, it will degenerate to Bit.

|  |  |
| --- | --- |
| Attribution | Description |
| value | The value of the bit, it only can be 0 or 1 |
| ids | The id of the bit. If the bit is generated by degenerating a qubit, the format of the id should be “qx”; otherwise, the format of the id should be “cx” |

|  |  |
| --- | --- |
| Methods | Description |
| \_\_init\_\_ | Init the instance, set the value and the ids. If the bit is generated by degenerating, then the parameter “ids” is required; otherwise, the parameter is optional. |
| \_\_add\_\_ | Overwrite the add of the instance, for example: bit+bit=str,0+1=01 |

Class baseQubit:

The parent class of Qubit and Qubits.

|  |  |
| --- | --- |
| Attribution | Description |
| matrix | The matrix of the instance, the format of this attribution is [[],[],[]] |
| amplitude | The amplitude of the instance, the format of this attribution is [,,,] |

Note: amplitude[i] is equal to matrix[i][0]

|  |  |
| --- | --- |
| Methods | Description |
| setAmp | Set the amplitude according to the matrix of this instance. |
| setMatrix | The parameter of the methods is a new matrix. Set the new matrix to self.matrix then call the method “setAmp”. |
| getMatrix | Get the matrix |
| getAmp | Get the amplitude |
| \_\_mul\_\_ | Overwrite the multiplication, compute the tenser product of two baseQubit |
| normalize | Normalize the matrix of the instance, then call setMatrix. The function was only called by Qubit.decideProb() |
| \_\_init\_\_ | Set the matrix and amplitude to empty list |

Class Qubit:

This class is inherited from class baseQubit. It is the basic computational unit and the class stands for a single-qubit.

|  |  |
| --- | --- |
| Attribution | Description |
| mode | This is a class attribution. It stands for the execute mode of this qubit. The value of this attribution can only be “smimulator” or “theory”. |
| matrix | The matrix of this qubit. And the format of it is [[],[]] 2\*1 |
| amplitude | The amplitude of this qubit. The format of it is [,] 1\*2 |
| assignmentError | The error rate of assignment. The value of it is get from errorRate.cfg according to self.ids if current mode is “simulator”, otherwise the value is 0. |
| singleGateError | The error rate of acting single-qubit gate on this qubit. The value of it is get from errorRate.cfg according to self.ids if current mode is “simulator”, otherwise the value is 0. |
| ids | The id of this qubit. And it can’t be repeated. |
| idList | Class attribution. It records the id of all the qubits in this experiment. |
| entanglement | This attribution is used to mark the Qubits of this qubit. If the qubit is in entanglement state, then this attribution records the Qubits instance; otherwise this attribution is None. |

|  |  |
| --- | --- |
| Methods | Description |
| setMatrix | Inherit from parent class baseQubit. |
| setAmp | Inherit from parent class baseQubit. |
| \_\_mul\_\_ | Inherit from parent class baseQubit. |
| \_\_init\_\_ | Initialize the qubit. |
| delete | Clean the entanglement of this qubit and delete it from the Qubit.idList |
| degenerate | Degenerate the qubit to bit. The method is called by M() |
| getMatrix | Overwrite the function of parent class baseQubit. If the qubit is not in entanglement state, then return self.matrix directly; if the qubit is in, call decideProb to compute the amplitude of this qubit and return the value. |
| getAmp | Overwrite the function of parent class baseQubit. If the qubit is not in entanglement state, then return self.amplitude; if the qubit is in, call decideProb to compute the amplitude of this qubit and return the value. |
| decideProb | The parameter of this method is a list of qubit. If the qubit is not in entangled state, just use self.amplitude to compute the probability. Otherwise, all the qubits in this list should be in self.entanglement. This method will compute the probability and the corresponding state of these qubits in the list. |
| recordQubit | Store the message of the qubit in Circuit instance. |

Note: Don’t get the matrix or amplitude of this qubit by self.matrix or self.amplitude, if you heve to get these information, please call slef.getAmp or self.getMatrix. Because once the qubit is in entanglement state, only the whole state of the Qubits has meaning for us except when talking about the measurement.

Class Qubits:

This class in inherited from class baseQubit. It contains more than two qubits. This class stands for the entangled state and the size of the matrix and amplitude is increasing exponentially.

|  |  |
| --- | --- |
| Attribution | Description |
| number | The attribution records the number of the entangled qubits. |
| matrix | The matrix of this qubits. |
| amplitude | The amplitude of this qubits. |
| qubitList | The attribution records the instance of each instance of the entangled qubits. |

|  |  |
| --- | --- |
| Methods | Description |
| \_\_init\_\_ | There are two parameters and they are all Qubit. Then compute the tensor product of these two qubits and use the product as the matrix of this instance. Please note, the two qubits should NOT be in entangled state, otherwise raise ValueError. |
| setMatrix | Inherit from parent class baseQubit. |
| setAmp | Inherit from parent class baseQubit. |
| getMatrix | Inherit from parent class baseQubit. |
| getAmp | Inherit from parent class baseQubit. |
| \_\_mul\_\_ | Inherit from parent class baseQubit. |
| \_\_getitem\_\_ | Enable us to get the qubit of this qubits by index, that is, qubits[i] to get the iTH qubit of this qubits |
| addNewItem | The type of the parameter can be Qubit or Qubits. Add this element to current Qubits instance. |
| mulMatrix | There are two parameters: two matrices, and the format of these two parameters are all [[],[],…]. This method will compute the tensor product of these matrices and return the result. |
| getIndex | The parameter is a qubit. Get the index of the qubit in current Qubits. If the qubit isn’t belong to this Qubits, return -1. |
| deleteItem | The parameter is a list of qubits. Delete these qubits from this instance and call Qubit.delete() to degenerate these qubits to bits. Then call qubit.decideProb to compute of amplitude and matrix of the qubits which are still in this Qubits. |

Class Circuit:

There is only one instance in each experiment.

|  |  |
| --- | --- |
| Attribution | Description |
| currentIDList | Class attribution. It is a list and records the id of all the Circuit instances. The environment is correct if there is only one element in this list and the element is equal to the ids of the current instance. |
| instance | Class attribution. We use this attribution to record the latest Circuit instance. And the newer instance will cover the original. |
| name | The name of this circuit. If the user give it as a parameter to \_\_init\_\_, the attribution will be set to this parameter; if the user don’t give the parameter, it will be set to “EXP”+current time automatically. |
| ids | It is the unique identifier of this circuit. |
| qubitExecuteList | This attribution stands for the execute process of each qubit. The type of this attribution is Dict. The key of the dictionary is the id of each qubit in this circuit, and the value of the dictionary is the execute process of each qubit. For example:  {q.ids: [“X q.ids;…”],  q.ids:[“ Y q.ids,…”]} |
| qubitNum | The total number of the qubits in this circuit |
| Urls | The url of the output files of this circuit |
| measureList | This attribution is used to record qubits which need to be measured. We should point out that the attribution is initialized in M() in Gate.py. |
| beginTime | The attribution records the begin time of the circuit execution. Get the current time by Datetime |
| endTime | The attribution records the termination time of the circuit execution. |
| beginMemory | The attribution records the begin memory of the circuit execution. Get the information of the memory by Psutil library. |
| endMemory | The attribution records the termination memory of the circuit execution. |
| mode | This attribution stands for the execute mode of this circuit which is got from executeMode.cfg. The value of this attribution only can be “simulator” or “theory” |

|  |  |
| --- | --- |
| Methods | Description |
| \_\_init\_\_ | Initialize the instance. There is only one parameter, namely “experimentName”, and the parameter is optional. And in this method, we create a new folder in “result”folder. |
| \_\_del\_\_ | Destructor. Delete the ids of current instance from currentIDList, which is a class attribution. |
| \_\_exportCircuit | Draw the circuit diagram according the qubitExecuteList |
| \_\_QASM | Translate the code to QASM code and store the QASM code in a txt file. We should point out that the process of translation is still developing. |
| execute | This method has only one parameter: executeTimes. It stands for the times of producting random numbers, the value of this parameter is larger, the result will be more precise. Please note that the measurement will actually execute only calling this method. |
| checkEnvironment | Judge whether there is only one Circuit instance. This method will return a Bool value. |
| \_\_exportChart | Draw the bar-chart according to the result of measurement. |
| \_\_orderTheID | This method use quick-sort to order the first parameter, namely idList, by asc and adjust the order of the second parameter, namely order, simultaneously. Because there may be entangled qubits in the measured qubits list, and the entangled qubits will disturb the order of measurement. Call this function will recover the order of the measured qubits. |
| \_\_countGate | This method is to figure out the number of gates used in this circuit. There are three types of gates:   1. single-qubit gate 2. double-qubit gate 3. measurement |
| \_\_removeQubit | The parameter is a list of qubits. Remove all elements of this list from this instance. |
| \_\_randomM | This method has two parameters: executive times and probability list. Product random numbers in [0, 1] for executive times and figure out the times it apparent in each interval. Then return the result. |
| \_\_printExecuteMsg | Print the executive message (that is, the number of qubits, the number of gate number, the executive time, the executive memory and so on) to command-line interface and the result.log. This method has four parameters: stateList, probList, gateNum and totalQubitNum |
| \_\_printPreMsg | Print the Pre-message(that is, the name of the circuit ) to command-line interface and the result.log |
| \_\_exportOriData | Export the original data (that is, the state and the corresponding the number of times it appeared in the result) to csv file. This method has two parameters: stateList and timesList. |

Class IBMQX

This class in used to call the api applied by IBM. So you should install the package “IBMQuantumExperience”. We should point out that you should replace the “token”in IBMToken.cfg by your own. You can get your own token in <https://quantumexperience.ng.bluemix.net/qx>. And the package is available in:

https://github.com/QISKit.

|  |  |
| --- | --- |
| Attribution | Description |
| \_\_config | The configuration information of the api. |
| api | The instance of IBMQuantumExperience class. We can use it to call the api applied by IBM. |
| device | The name of the device which is used as the experimental platform. The value of this attribution is got from IBMToken.cfg. |
| shot | Executive times. The value of this attribution is got from IBMToken.cfg. |
| connectivity | The type of this attribution is Dict. It stands for the connectivity map of current device. For example, {0: [1,2]} means that if we use q0 as control-qubit, then we can choose q1 or q2 as target-qubit. Please note that the type of each element in this Dict should be int, not string. |

|  |  |
| --- | --- |
| Methods | Description |
| \_\_init\_\_ | Initialize this instance. |
| \_\_getAvailalbeBak | Call api.available\_backends() to get the available devices. |
| executeQASM | Execute the QASM code in IBM device. The code is got from QASM.txt. |
| \_\_canExecute | Adjust the QASM code so that it satisfies the requirement of the connectivity map of the current device. But this adjustment won’t always be successful. If successful, this method will return the new QASM code; if failed, return None. We should point out that this method is still under development. |
| \_\_getTotalConnectivity | Set the self.connectivity and the reverse\_cnot to a new Dict and return the dict. |
| \_\_determindID | Judge whether the number of qubits in this circuit is more than the actual number of qubits in this IBM chip. If bigger, return False; else return True. And adjust the ID of the qubit to satisfy the requirement of the actual device. For example, the if of the qubit in this circuit is 10, 20, but only 0,1,2,3,4 is allowed in IBM device. Then adjust 10 and 20 to 0 and 1. |
| \_\_adjustCNOT | Check the CNOT list whether satisfies the constraint of the totalConnectivity. |
| \_\_backTrace | Searching the solution space by backtracing. |
| \_\_getQubitMap | There three parameters: l1,l2 and totalConnectivity. Use the element in l1 as key and the corresponding element in l2 as value to construct a new Dict. If there are some qubit not in CNOTQList, the function will append the qubit in the tail of the new Dict |
| \_\_checkMapConstraint | There are two parameters: map and totalConnectivity. The function will adjust the copy of the CNOTList according to the map and call \_\_checkAllConstraint to judge whether the new map is allowed in IBM chip. |
| \_\_changeQASMandCNOT | There is only one parameter:map. Adjust the QASM code, CNOTList and qubitList according to the map. Note that QASM, CNOTList and qubitList are all global variables and stated in \_\_canExecute. |
| \_\_checkSingleConstraint | There are two arguments: cnot and totalConnectivity. Judge whether the cnot is allowed in totalConnectivity. |
| \_\_checkAllConstraint | There are two arguments: cnotList and totalConnectivity. Call \_\_checkSingleConstraint for len(cnotList) times, and judge whether the elements of cnotList are all in totalConnectivity. |
| \_\_reverseCNOT | Reverse the CNOT gate. For example, if (Qx:Qy) is in QASM, but (Qy,Qx) is allowed in IBM chip, then add H gate to reverse the control-qubit and the target-qubit |