QiskiFT

Release 1.0.0

Owen Dugan

May 16, 2021

CONTENTS:

QiskiFT is a package for implementing Quantum Error Correction and Quantum Fault Tolerance in Python using Qiskit. It automates much of the process of implementing fault tolerant computation, allowing users to create fault-tolerant circuits in only a few more lines of code than non-fault-tolerant circuits. For example, Deutsch's Algorithm can be implemented fault-tolerantly in 15 lines of code.

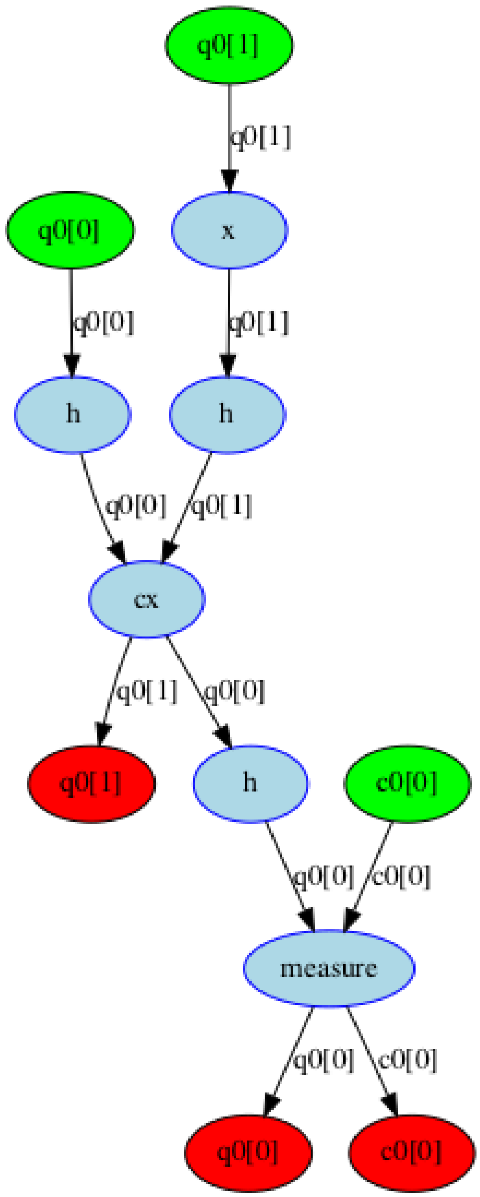


Fig. 1: A non-fault-tolerant implementation of Deutsch's Algorithm.

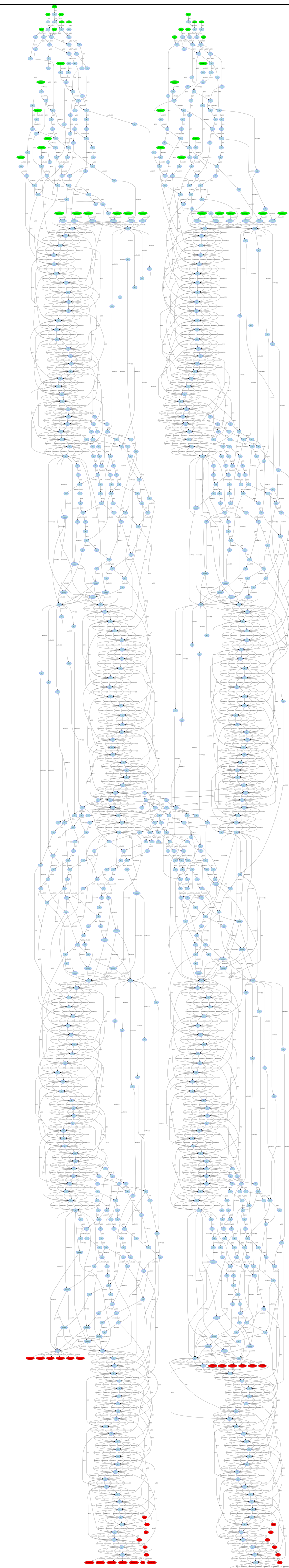


Fig. 2: A fault-tolerant implementation of Deutsch's Algorithm.

CHAPTER

ONE

QISKIFT API

1.1 The BaseFaultTolerance Module

The BaseFaultTolerance module contains base classes for Quantum Error Correction and Quantum Fault Tolerance. These classes are generic; they require the user to provide the relevant algorithms when they are initialized. For specific quantum codes, see the Codes page. Note that only the Steane code is currently implemented.

`class BaseFaultTolerance.BaseFaultTolerantMeasurement(*args, **kwargs)`

A class for implementing fault-tolerant measurement. NOT YET IMPLEMENTED.

Attributes

`is_analysis_pass` Check if the pass is an analysis pass.

`is_transformation_pass` Check if the pass is a transformation pass.

Methods

`name`()	Return the name of the pass.
`run`(dag)	Run a pass on the DAGCircuit.

`class BaseFaultTolerance.Encoder(encoderCircuit, numAncillas)`

A class for implementing the non-fault tolerant ecoding of the Steane $|0\rangle$ state.

Methods

createEncoderCircuit :	Creates a circuit encoding the $|0\rangle$ state
createEncoderDag :	Creates a DAG encoding the $|0\rangle$ state
getEncoderCircuit :	Adds gates encoding the $|0\rangle$ state to a circuit
getEncoderDag :	Adds gates encoding the $|0\rangle$ state to a DAG

`createEncoderCircuit(numQubits)`

Creates a circuit encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`createEncoderDag(numQubits)`

Creates a DAG encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`getEncoderCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)

Encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given circuit.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`getEncoderDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)

Encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given DAG.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`class BaseFaultTolerance.ErrorCorrector`(*syndromeDetector*, *syndromeCorrector*)

A class for implementing non-fault tolerant error correction (syndrome detection and correction) for an arbitrary error correction scheme. This class combines `SyndromeDetection` and `SyndromeCorrection` into a single class for ease of use.

Parameters

syndromeDetector [SyndromeDetector] An object representing syndrome detection.

syndromeCorrector [SyndromeCorrector] An object representing syndrome correction.

Methods

errorCorrectCircuit :	Implements error correction for the given circuit.
errorCorrecDag :	Implements error correction for the given DAG.

`errorCorrectCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)

Creates gates implementing fault tolerant error correction for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform error correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform error correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform error correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`errorCorrectDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)
Creates gates implementing non-fault tolerant error correction for the given qubits in the given DAG.

Parameters

dag [DAGCircuit] The dag for which to perform error correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform error correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform error correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`class BaseFaultTolerance.`**`FaultTolerance`**(**args*, ***kwargs*)
A Transpiler pass that converts a given quantum computation into an equivalent one with error correction. NOT YET IMPLEMENTED

Attributes

`is_analysis_pass` Check if the pass is an analysis pass.

`is_transformation_pass` Check if the pass is a transformation pass.

Methods

`name()`	Return the name of the pass.
`run``()`	Run a pass on the DAGCircuit.

`run`()
Run a pass on the DAGCircuit. This is implemented by the pass developer.

Args: dag (DAGCircuit): the dag on which the pass is run.

Raises: NotImplementedError: when this is left unimplemented for a pass.

`class BaseFaultTolerance.`**`FaultTolerantEncoder`**(*encoder*, *checkerCircuit*, *numAncillas*, *correctVal*, *numRepeats*)
A class for implementing an fault tolerant ecoding of the $|0\rangle$ state for an arbitrary quantum code.

Parameters

encoder [Encoder] An Encoder object representing the $|0\rangle$ state non-fault tolerant encoding process.

checkerCircuit [QuantumCircuit] A circuit for determining whether the $|0\rangle$ state has been encoded properly.

numAncillas [int] The number of ancilla qubits used to check the encoded effect. Note: the ancilla qubits must be at the end of the list of qubits for the circuit.

correctVal [int] The classical register value corresponding to the correct initialization of the encoded $|0\rangle$ state.

numRepeats [int] The number of times to attempt to create the encoded $|0\rangle$ state.

Methods

createEncoderCircuit :	Creates a circuit encoding the $|0\rangle$ state
createEncoderDag :	Creates a DAG encoding the $|0\rangle$ state
getEncoderCircuit :	Adds gates encoding the $|0\rangle$ state to a circuit
getEncoderDag :	Adds gates encoding the $|0\rangle$ state to a DAG

`createEncoderCircuit`(*numQubits*)

Creates a circuit fault-tolerantly encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`createEncoderDag`(*numQubits*)

Creates a DAG fault-tolerantly encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`getEncoderCircuit`(*circuit*, *qregs*, *cregs1=None*, *ancillas1=None*, *cregs2=None*, *ancillas2=None*)

Fault-tolerantly encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given circuit.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`getEncoderDag`(*dag*, *qregs*, *cregs1=None*, *ancillas1=None*, *cregs2=None*, *ancillas2=None*)

Fault-tolerantly encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given DAG.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`class BaseFaultTolerance.`**`FaultTolerantGates`**`(`*gatesToCircuit*`)`

A class for implementing fault tolerant gates for an arbitrary quantum error correction code.

Parameters

gatesToCircuit [map(str, (QuantumCircuit, int))] A map representing conversions between gates and circuits implementing fault tolerant versions of those gates. The keys of the map are the QASM label for the gate in question, given by `gate.qasm()`. The outputs of the map are tuples of the form `(circuit, numAncillas)`, where `circuit` is a fault-tolerant implementation of a gate and `numAncillas` is the number of ancillas qubits used in the fault-tolerant implementation of the gate.

Methods

addGateCircuit :	Adds a fault tolerant gate to the given circuit.
addGateDag :	Adds a fault tolerant gate to the given DAG.

`addGateCircuit``(`*circuit*, *gate*, *qregs*, *cregs=None*, *ancillas=None*`)`

Adds the specified number of fault tolerant implementations of a quantum gate to the given circuit.

Parameters

circuit [QuantumCircuit] The circuit on which to perform the fault tolerant gate.

gate [Gate] The non-fault tolerant gate for which to implement a fault tolerant version.

qregs [list(list(QuantumRegister))] The Quantum Registers to on which to perform the fault tolerant gate. Each `qregs[i]` represents the list of quantum registers which correspond to the ith input to the non-fault tolerant version of the gate in question. Note that each `qregs[i]` must have the same length.

cregs [list(list(ClassicalRegister)), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `cregs[j]` classical register.

ancillas [list(list(AncillaRegister)), list(list(QuantumRegister)), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `ancillas[j]` ancilla register.

`addGateDag``(`*dag*, *gate*, *qregs*, *cregs=None*, *ancillas=None*`)`

Adds the specified number of fault tolerant implementations of a quantum gate to the given DAG.

Parameters

dag [DAGCircuit] The dag on which to perform the fault tolerant gate.

gate [Gate] The non-fault tolerant gate for which to implement a fault tolerant version.

qregs [list(list(QuantumRegister))] The Quantum Registers to on which to perform the fault tolerant gate. Each `qregs[i]` represents the list of quantum registers which correspond to the ith input to the non-fault tolerant version of the gate in question. Note that each `qregs[i]` must have the same length.

cregs [list(list(ClassicalRegister)), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `cregs[j]` classical register.

ancillas [list(list(AncillaRegister)), list(list(QuantumRegister)), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `ancillas[j]` ancilla register.

`class BaseFaultTolerance.`**`SyndromeCorrector`**`(`*correctorCircuit*`)`

A class for implementing fault tolerant syndrome correction for an arbitrary error correction scheme.

Parameters

correctorCircuit [QuantumCircuit] A Quantum Circuit implementing fault tolerant syndrome correction for a single qubit.

Methods

syndromeCorrectCircuit :	Implements syndrome correction for the given circuit.
syndromeCorrectDag :	Implements syndrome correction for the given DAG.

`syndromeCorrectCircuit``(`*circuit*, *qregs*, *cregs*`)`

Creates gates implementing fault tolerant syndrome correction for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform syndrome correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform syndrome correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`syndromeCorrectDag``(`*dag*, *qregs*, *cregs*`)`

Creates gates implementing fault tolerant syndrome correction for the given qubits in the given DAG.

Parameters

dag [DAGCircuit] The dag for which to perform syndrome correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform syndrome correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`class BaseFaultTolerance.`**`SyndromeDetector`**`(`*detectorCircuit*, *numAncillas*`)`

A class for implementing non-fault tolerant syndrome detection for an arbitrary error correction scheme.

Parameters

detectorCircuit [QuantumCircuit] A Quantum Circuit implementing non-fault tolerant syndrome detection.

numAncillas [int] The number of ancilla qubits used in the syndrome detection.

Methods

syndromeDetectCircuit :	Implements syndrome detection for the given circuit.
syndromeDetectDag :	Implements syndrome detection for the given DAG.

`syndromeDetectCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)
Creates gates implementing non-fault tolerant syndrome detection for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform syndrome detection.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome detection.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`syndromeDetectDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)
Creates gates implementing non-fault tolerant syndrome detection for the given qubits in the given DAG.

Parameters

dag [DAGCircuit] The DAG for which to perform syndrome detection.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome detection.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

1.2 The Steane Module

The Steane Module implements Quantum Error Correction and Quantum Fault Tolerance using the Steane 7-qubit code. The 7-qubit code encodes the state $|\phi\rangle$ as

$$|\tilde{\phi}\rangle = (1 + X_0X_4X_5X_6)(1 + X_1X_3X_5X_6)(1 + X_2X_3X_4X_6)|\phi\rangle.$$

For Syndrome Detection, the Steane code measures 6 operators:

$$M_a = X_0X_4X_5X_6,$$

$$M_b = X_1X_3X_5X_6,$$

$$M_c = X_2X_3X_4X_6,$$

$$N_a = Z_0Z_4Z_5Z_6,$$

$$N_b = Z_1Z_3Z_5Z_6,$$

and

$$N_c = Z_2Z_3Z_4Z_6.$$

More details about each aspect of the Steane code are provided below.

`class Steane.SteaneEncoder`

Bases: *BaseFaultTolerance.Encoder*

A class for implementing non-fault tolerant preparation of the Steane $|0\rangle$ state. As described at the top of this page, the $|0\rangle$ state is encoded as

$$|\tilde{0}\rangle = (1 + X_0X_4X_5X_6)(1 + X_1X_3X_5X_6)(1 + X_2X_3X_4X_6)|0\rangle.$$

The circuit representation of the initialization process is:

Methods

createEncoderCircuit :	Creates a circuit encoding the $\|0\rangle$ state
createEncoderDag :	Creates a DAG encoding the $\|0\rangle$ state
getEncoderCircuit :	Adds gates encoding the $\|0\rangle$ state to a circuit
getEncoderDag :	Adds gates encoding the $\|0\rangle$ state to a DAG

`createEncoderCircuit`(*numQubits*)

Creates a circuit encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`createEncoderDag`(*numQubits*)

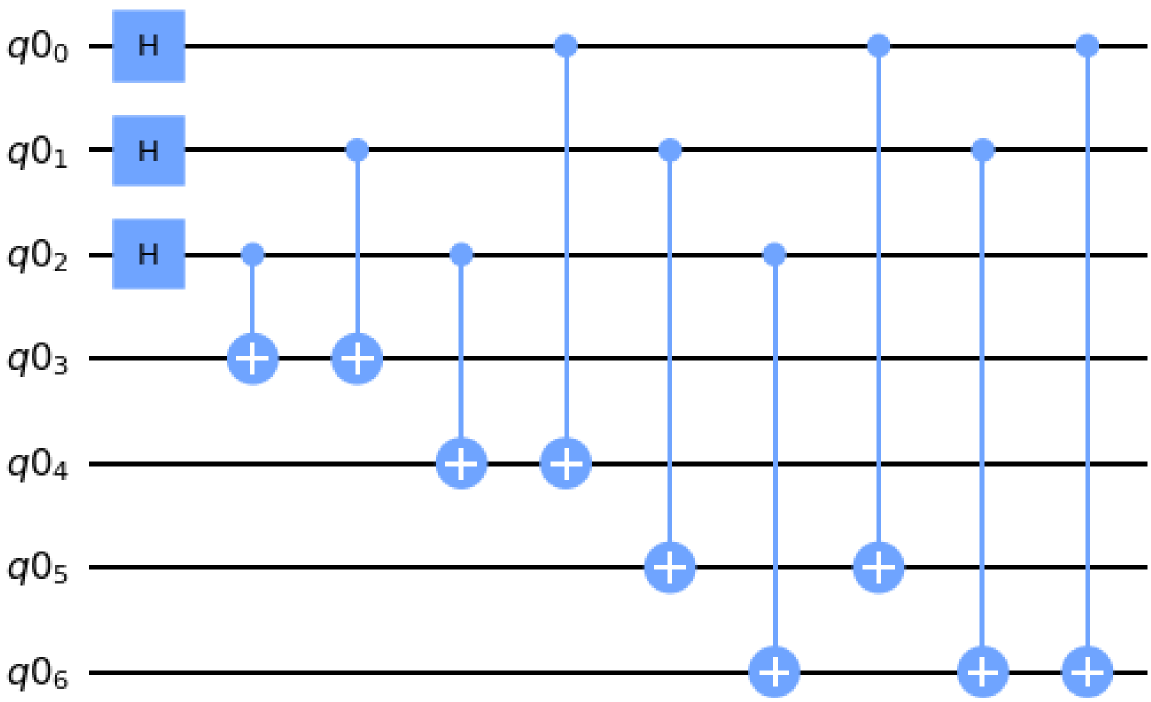
Creates a DAG encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`getEncoderCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)

Encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given circuit.



Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`getEncoderDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)

Encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given DAG.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

class `Steane.`**`SteaneErrorCorrector`**

Bases: *`BaseFaultTolerance.ErrorCorrector`*

A class for implementing non-fault tolerant error correction for the Steane Code. This class combines `SteaneSyndromeDetection` and `SteaneSyndromeCorrection` into a single class for ease of use.

Methods

errorCorrectCircuit :	Implements error correction for the given circuit.
errorCorrecDag :	Implements error correction for the given DAG.

`errorCorrectCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)

Creates gates implementing fault tolerant error correction for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform error correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform error correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform error correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`errorCorrectDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)

Creates gates implementing non-fault tolerant error correction for the given qubits in the given DAG.

Parameters

dag [DAGCircuit] The dag for which to perform error correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform error correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform error correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

class `Steane.`**`SteaneFaultTolerantEncoder`**(*numRepeats*)

Bases: *`BaseFaultTolerance.FaultTolerantEncoder`*

A class for implementing fault tolerant ecoding of the Steane encoded $|0\rangle$ state. NOT FINISHED.

Parameters

numRepeats [int] The number of times to try to create the $|0\rangle$ state before giving up.

Methods

createEncoderCircuit :	Creates a circuit encoding the $\|0\rangle$ state
createEncoderDag :	Creates a DAG encoding the $\|0\rangle$ state
getEncoderCircuit :	Adds gates encoding the $\|0\rangle$ state to a circuit
getEncoderDag :	Adds gates encoding the $\|0\rangle$ state to a DAG

`createEncoderCircuit`(*numQubits*)
Creates a circuit fault-tolerantly encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`createEncoderDag`(*numQubits*)
Creates a DAG fault-tolerantly encoding the specified number of qubits to the encoded $|0\rangle$ state.

Parameters

numQubits [int] The number of qubits to initialize to the encoded $|0\rangle$ state.

`getEncoderCircuit`(*circuit*, *qregs*, *cregs1=None*, *ancillas1=None*, *cregs2=None*, *ancillas2=None*)
Fault-tolerantly encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given circuit.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`getEncoderDag`(*dag*, *qregs*, *cregs1=None*, *ancillas1=None*, *cregs2=None*, *ancillas2=None*)
Fault-tolerantly encodes the specified Quantum Registers to the encoded $|0\rangle$ state for the given DAG.

Parameters

dag [DAGCircuit] The circuit for which to create the encoding.

qregs [list(QuantumRegister)] The Quantum Registers to encode to the $|0\rangle$.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to encode to the $|0\rangle$, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to encode to the $|0\rangle$, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the encoding process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`class Steane.SteaneFaultTolerantGates`
Bases: `BaseFaultTolerance.FaultTolerantGates`

A class for implementing fault tolerant gates for the Steane Code. The current implemented gates are X, H, S, and CNOT. These gates can all be implemented bitwise. The figures below show the implementations for these four gates.

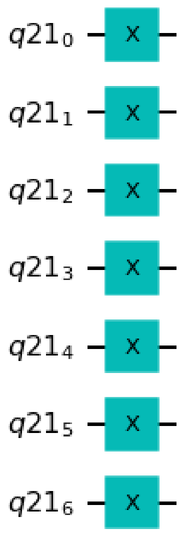


Fig. 1: The fault tolerant X gate.

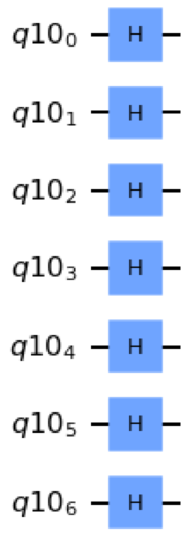


Fig. 2: The fault tolerant H gate.

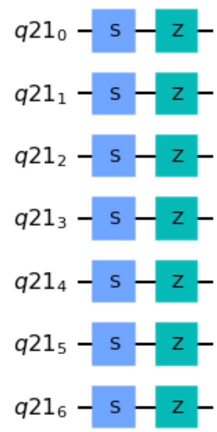


Fig. 3: The fault tolerant S gate.

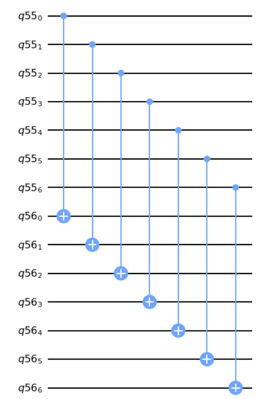


Fig. 4: The fault tolerant CNOT gate.

Methods

addGateCircuit :	Adds a fault tolerant gate to the given circuit.
addGateDag :	Adds a fault tolerant gate to the given DAG.

addGateCircuit(*circuit*, *gate*, *qregs*, *cregs=None*, *ancillas=None*)

Adds the specified number of fault tolerant implementations of a quantum gate to the given circuit.

Parameters

circuit [QuantumCircuit] The circuit on which to perform the fault tolerant gate.

gate [Gate] The non-fault tolerant gate for which to implement a fault tolerant version.

qregs [list(list(QuantumRegister))] The Quantum Registers to on which to perform the fault tolerant gate. Each `qregs[i]` represents the list of quantum registers which correspond to the ith input to the non-fault tolerant version of the gate in question. Note that each `qregs[i]` must have the same length.

cregs [list(list(ClassicalRegister)), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `cregs[j]` classical register.

ancillas [list(list(AncillaRegister)), list(list(QuantumRegister)), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `ancillas[j]` ancilla register.

`addGateDag`(*dag*, *gate*, *qregs*, *cregs=None*, *ancillas=None*)

Adds the specified number of fault tolerant implementations of a quantum gate to the given DAG.

Parameters

dag [DAGCircuit] The dag on which to perform the fault tolerant gate.

gate [Gate] The non-fault tolerant gate for which to implement a fault tolerant version.

qregs [list(list(QuantumRegister))] The Quantum Registers to on which to perform the fault tolerant gate. Each `qregs[i]` represents the list of quantum registers which correspond to the ith input to the non-fault tolerant version of the gate in question. Note that each `qregs[i]` must have the same length.

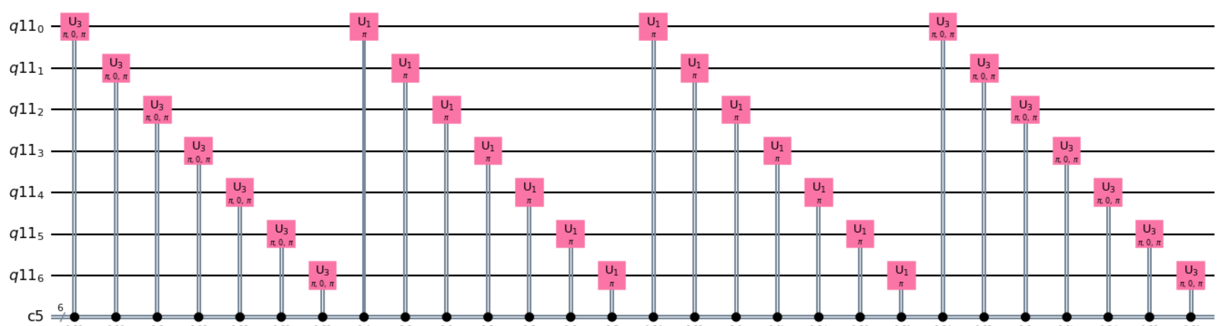
cregs [list(list(ClassicalRegister)), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `cregs[j]` classical register.

ancillas [list(list(AncillaRegister)), list(list(QuantumRegister)), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(qregs[0])` and the syndrome detection process for the `qregs[i][j]` quantum register will use the `ancillas[j]` ancilla register.

class `Steane.SteaneSyndromeCorrector`

Bases: *BaseFaultTolerance.SyndromeCorrector*

A class for implementing fault tolerant syndrome correction for the Steane code. The circuit representation for Syndrome Correction is shown below:



Methods

syndromeCorrectCircuit :	Implements syndrome correction for the given circuit.
syndromeCorrectDag :	Implements syndrome correction for the given DAG.

`syndromeCorrectCircuit`(*circuit*, *qregs*, *cregs*)

Creates gates implementing fault tolerant syndrome correction for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform syndrome correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform syndrome correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

`syndromeCorrectDag`(*dag*, *qregs*, *cregs*)
Creates gates implementing fault tolerant syndrome correction for the given qubits in the given DAG.

Parameters

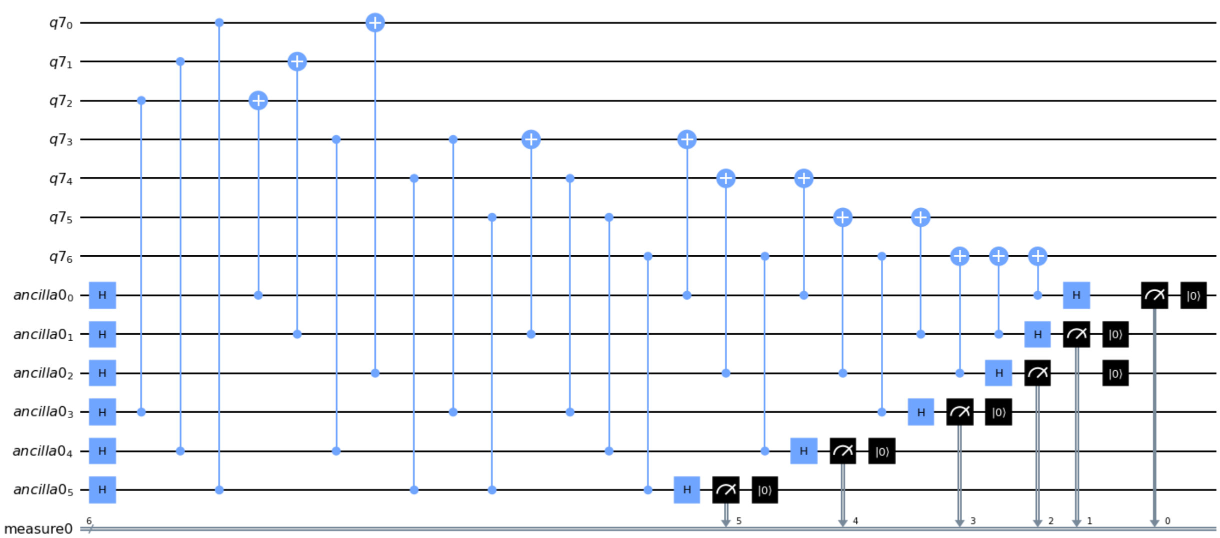
dag [DAGCircuit] The dag for which to perform syndrome correction.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome correction.

cregs [list(ClassicalRegister)] The Classical Registers used to perform syndrome correction, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome correction process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

class `Steane.SteaneSyndromeDetector`
Bases: *BaseFaultTolerance.SyndromeDetector*

A class for implementing non-fault tolerant syndrome detection for the Steane Code. Syndrome detection works by measuring six stabilizer operators, M_a, M_b, M_c, N_a, N_b, and N_c, defined at the top of this page. The circuit representation of the syndrome detection process is:



Methods

syndromeDetectCircuit :	Implements syndrome detection for the given circuit.
syndromeDetectDag :	Implements syndrome detection for the given DAG.

`syndromeDetectCircuit`(*circuit*, *qregs*, *cregs=None*, *ancillas=None*)
Creates gates implementing non-fault tolerant syndrome detection for the given qubits in the given circuit.

Parameters

circuit [QuantumCircuit] The circuit for which to perform syndrome detection.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome detection.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

`syndromeDetectDag`(*dag*, *qregs*, *cregs=None*, *ancillas=None*)
Creates gates implementing non-fault tolerant syndrome detection for the given qubits in the given DAG.

Parameters

dag [DAGCircuit] The DAG for which to perform syndrome detection.

qregs [list(QuantumRegister)] The Quantum Registers to on which to perform syndrome detection.

cregs [list(ClassicalRegister), Optional] The Classical Registers used to perform syndrome detection, if classical registers are needed. If `cregs` is provided, it must satisfy `len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `cregs[i]` classical register.

ancillas [list(AncillaRegister), list(QuantumRegister), Optional] The Ancilla Registers used to perform syndrome detection,, if ancilla registers are needed. If `ancillas` is provided, it must satisfy `len(ancillas) == len(cregs) == len(qregs)` and the syndrome detection process for the `qregs[i]` quantum register will use the `ancillas[i]` ancilla register.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX