QuaEC Documentation

Release 1.0.1

Chris Granade and Ben Criger

Jul 26, 2018

Contents

CHAPTER 1

Contents

1.1 Pauli and Clifford Groups

1.1.1 `qecc.Pauli`: Class representing Pauli group elements

The class *qecc.Pauli* is used to represent elements of the Pauli group $\mathcal{P}_n$ on n qubits. Instances can be constructed by specifying strings of `I`, `X`, `Y` and `Z`, corresponding to the specification of an operator in the Pauli group.

```
>>> import qecc as q
>>> P = q.Pauli('X')
>>> print P
i^0 X
>>> Q = q.Pauli('XZZXI')
>>> print Q
i^0 XZZXI
>>> R = q.Pauli('XYZ')
>>> print R
i^0 XYZ
```

Additionaly, a phase can be provided. Since only integer powers of i are allowed as phases, the phase of a *qecc.Pauli* instance is represented by an integer in `range(4)`. Any other integer is converted to an integer in that range that is equivalent mod 4.

```
>>> print q.Pauli('X', 2)
i^2 X
```

The *qecc.Pauli* class supports multiplication, tensor products and negation by the `*`, `&` and `-` operators, respectively.

```
>>> import qecc
>>> P = qecc.Pauli('X')
>>> Q = qecc.Pauli('Y')
>>> P * Q
```

(continues on next page)

(continued from previous page)

```
i^1 Z
>>> P & Q
i^0 XY
>>> -P * Q
i^3 Z
```

Using these operators, it is straightforward to construct instances of `qecc.Pauli` from existing instances. To make this easier, QuaEC provides single-qubit operators `I`, `X`, `Y` and `Z`.

```
>>> from qecc import I, X, Y, Z
>>> print q.Pauli('XZZXI') & I
i^0 XZZXII
```

Additionally, instances of `qecc.Pauli` can be tested for equality.

```
>>> -P * Q == P * -Q
True
>>> P * Q != Q * P
True
```

The length of a `qecc.Pauli` is defined as the number of qubits it acts upon.

```
>>> print len(qecc.Pauli('XYZI'))
4
```

This information is also exposed as the property `nq`.

```
>>> print qecc.Pauli('XYZI').nq
4
```

Class Reference

class qecc.**Pauli**(*operator*, *phase=0*)

Class representing an element of the Pauli group on n qubits.

Parameters

- **operator** (*str*) – String of I's, X's, Y's and Z's.
- **phase** (*int*) – A phase input as an integer from 0 to 3, interpreted as i^{phase}.

nq

Returns the number of qubits upon which this Pauli operator acts.

wt

Measures the weight of a given Pauli.

Return type int (between 0 and the number of qubits on which the Pauli is defined)

Returns The number of qubits on which the represented Pauli operator is supported.

str_sparse(*incl_ph=True*)

Returns a compact representation for `qecc.Pauli` objects, for those having support on a small number of qubits of a large register.

tens(*other*)

Concatenates the op strings of two Paulis, and multiplies their phases, to produce the Kronecker product of the two.

Parameters **other** (qecc.Pauli) – Pauli operator Q to be tensored with this instance.

Returns An instance representing $P \otimes Q$, where P is the Pauli operator represented by this instance.

set_phase (*ph=0*)

Returns a *qecc.Pauli* object having the same operator as the input, with a specified phase (usually used to erase phases).

mul_phase (*ph*)

Increments the phase of this Pauli by i^{ph}.

Parameters **ph** (*int*) – Amount the phase is to be incremented by.

Returns This instance.

permute_op (*perm*)

Returns a new *qecc.Pauli* instance whose operator part is related to the operator part of this Pauli, so that σ_μ is mapped to $\sigma_{\pi(\mu)}$ for some permutation π of the objects X, Y, Z.

For example:

```
>>> import qecc as q
>>> P = q.Pauli('XYZXYZ')
>>> print P.permute_op('ZXY')
i^0 ZXYZXY
```

Note that the result is **not** guaranteed to be the result of a Clifford operator acting on this Pauli, as permutation may not respect the phases introduced by taking products. For example:

```
>>> import qecc as q
>>> P = q.Pauli('XYZ')
>>> Q = q.Pauli('YYZ')
>>> P * Q
i^1 ZII
>>> Pp = P.permute_op('ZYX')
>>> Qp = Q.permute_op('ZYX')
>>> Pp * Qp
i^3 XII
```

Parameters **perm** (*list*) – A list indicating which permutation is to be performed.

Returns A new instance Q of *qecc.Pauli* that is related to this instance by a permutation of X, Y and Z.

as_gens ()

Expresses an input Pauli in terms of the elementary generators X_j and Z_j, stripping off phases.

Return type list of *qecc.Pauli* instances.

as_bsv ()

Converts the given Pauli to a binary symplectic vector, discarding phase information.

Returns A binary symplectic vector representing this Pauli operator.

Return type *BinarySymplecticVector*

as_circuit ()

Transforms an n-qubit Pauli to a serial circuit on n qubits. Neglects global phases.

Return type *qecc.Circuit*

as_unitary()

Returns a numpy.ndarray containing a unitary matrix representation of this Pauli operator.

Raises a RuntimeError if NumPy cannot be imported.

as_clifford()

Converts a Pauli into a Clifford which changes the signs of input Paulis. :returns: A Clifford representing conjugation by this Pauli operator. :rtype: *qecc.Clifford*

static from_sparse(*sparse_pauli*, *nq=None*)

Given a dictionary from non-negative integers to single-qubit Pauli operators or strings representing single-qubit Pauli operators, creates a new instance of *qecc.Pauli* representing the input.

```
>>> from qecc import Pauli, X, Y, Z
>>> print Pauli.from_sparse({3: X, 5: X, 7: Z}, nq=12)
i^0 X[3] X[5] Z[7]
```

Parameters

- **sparse_pauli** (*dict*) – Dictionary from qubit indices (non-negative integers) to single-qubit Pauli operators or to strings.
- **nq** (*int*) – If not None, specifies the number of qubits on which the newly created Pauli operator is to act.

static from_clifford(*cliff_in*)

Tests an input Clifford cliff_in to determine if it is, in fact, a Pauli. If so, it outputs the Pauli. If not, it raises an error.

Parameters **cliff_in** – Representation of Clifford operator to be converted, if possible.

Return type *qecc.Pauli*

Example:

```
>>> import qecc as q
>>> cliff = q.Clifford([q.Pauli('XI',2),q.Pauli('IX')], map(q.Pauli,['ZI','IZ
↪']))
>>> q.Pauli.from_clifford(cliff)
i^0 ZI
```

Converting a Pauli into a Clifford and back again will erase the phase:

```
>>> import qecc as q
>>> paul = q.Pauli('YZ',3)
>>> cliff = paul.as_clifford()
>>> q.Pauli.from_clifford(cliff)
i^0 YZ
```

static from_string(*bitstring*, *p_1*)

Creates a multi-qubit Pauli using a bitstring and a one-qubit Pauli, by replacing all instances of 1 in the bitstring with the appropriate Pauli, and replacing all instances of 0 with the identity.

Parameters

- **bitstring** (*list*) – a list of integers, each of which is either 0 or 1. *bitstring* can also be a string, type conversion is automatic.
- **p_1** (*str*) – a single-qubit Pauli.

Returns a phaseless Pauli from a bitstring. The intended use of this function is as a quick means of specifying binary Paulis. p_1 is the one_qubit Pauli that a '1' represents.

Return type `qecc.Pauli`

Example:

```
>>> import qecc as q
>>> bitstring = '101110111100'
>>> p_1 = q.Pauli('X')
>>> q.Pauli.from_string(bitstring, p_1)
i^0 XIXXXIXXXXII
```

reg_wt (***kwargs*)

Produces the number of qubits within a subset of the register on which the Pauli in question acts non-trivially.

Parameters **region** (*tuple*) – a tuple containing the indices on which the weight is to be evaluated.

Returns the number of qubits in the sub-register on which the Pauli `self` does not act as the identity.

cust_wt (*char*)

Produces the number of qubits on which an input Pauli acts as a specified single-qubit Pauli.

Parameters **char** (*str*) – a single-letter string containing an I, X, Y or Z.

Returns the number of qubits in the Pauli `self` which are acted upon by the single-qubit operator `char`.

ct ()

The conjugate transpose of this Pauli operator.

Return type an instance of the `qecc.Pauli` class.

centralizer_gens (*group_gens=None*)

Returns the generators of the centralizer group $\mathrm{C}(P)$, where P is the Pauli operator represented by this instance. If `group_gens` is specified, $\mathrm{C}(P)$ is taken to be a subgroup of the group $G = \langle G_1, \ldots, G_k \rangle$, where G_i is the i^{th} element of `group_gens`.

Parameters **group_gens** (list of `qecc.Pauli` instances) – Either `None` or a list of generators G_i. If not `None`, the returned centralizer $\mathrm{C}(P)$ is a subgroup of the group $\langle G_i \rangle_{i=1}^k$.

Returns A list of elements P_i of the Pauli group such that $\mathrm{C}(P) = \langle P_i \rangle_{i=1}^n$, where n is the number of unique generators of the centralizer.

hamming_dist (*other*)

Returns the Hamming distance between this and another Pauli operator, defined as $d(P, Q) = \mathrm{wt}(PQ)$.

Iterating Over Groups and Subgroups

qecc.**pauli_group** (*nq*)

Generates an iterator onto the Pauli group of n qubits, where n is given as the argument *nq*.

Parameters **nq** (*int*) – The number of qubits acted upon by the returned Pauli group.

Returns An iterator such that `list(pauli_group(nq))` produces a list of all possible Pauli operators on `nq` qubits.

qecc.**from_generators**(*gens*, *coset_rep=None*, *incl_identity=True*)

Given a list of generators `gens`, yields an iterator onto the group generated by products of elements from `gens`.

If `coset_rep` is specified, returns the coset of the group generated by `gens` represented by `coset_rep`.

Utility Functions

qecc.**com**(*P*, *Q*)

Given two elements *P* and *Q* of a Pauli group, returns 0 if $[P, Q] = 0$ and returns 1 if $\{P, Q\} = 0$.

Parameters

- **P** (`qecc.Pauli`) – Representation of P.
- **Q** (`qecc.Pauli`) – Representation of Q.

Returns $c(P, Q)$.

Return type `int`

qecc.**elem_gens**(*nq*)

Produces all weight-one X and Z operators on *nq* qubits. For example,

```
>>> import qecc as q
>>> Xgens, Zgens = q.elem_gens(2)
>>> print Xgens[1]
i^0 IX
```

Parameters **nq** (*int*) – Number of qubits for each returned operator.

Returns a tuple of two lists, containing X and Z generators, respectively.

qecc.**eye_p**(*nq*)

Given a number of qubits, returns the identity Pauli on that many qubits.

Parameters **nq** (*int*) – Number of qubits upon which the returned Pauli acts.

Return type *qecc.Pauli*

Returns A Pauli operator acting as the identity on each of `nq` qubits.

Searching Over Pauli Group Elements

QuaEC provides useful tools for searching over elements of the Pauli group. A few particlar searches are provided built-in, while other searches can be efficiently built using the predicates described in *Predicates and Filters*.

qecc.**is_in_normalizer**(*pauli*, *stab*)

Given an element `pauli` of a Pauli group and the generators `stab` of a stabilizer group S, returns True if and only if `pauli` is in the normalizer $N(S)$.

qecc.**mutually_commuting_sets**(*n_elems*, *n_bits=None*, *group_gens=None*, *exclude=None*)

Yields an iterator onto tuples representing mutually commuting sets of `n_elems` independent Pauli operators, excluding the identity Pauli.

Parameters

- **n_elems** (*int*) – The number of mutually commuting Pauli operators to include in each tuple.

- **n_bits** (*int*) – The number of qubits on which each Pauli operator considered by this iterator acts. If None, defaults to the number of qubits on which the first element of group_gens acts.
- **group_gens** (None or a sequence of *qecc.Pauli* instances) – The generators of the group in which to search for mutually commuting Pauli operators. Defaults to the elementary generators of the Pauli group on n_bits qubits.
- **exclude** (None or a sequence of *qecc.Pauli* instances) – If not None, the iterator will omit from its search any operators in the group generated by exclude.

1.1.2 qecc.Clifford: Class representing Clifford group elements

Elements of the automorphism group of the Pauli group (known as the Clifford group) are represented by the class *qecc.Clifford*. Instances of Clifford are constructed by specifying the mappings of the generators of the Pauli group, such that the action of a Clifford instance is defined for all input Pauli group elements.

```
>>> import qecc as q
>>> C = q.Clifford(['XX', 'IX'], ['ZI', 'ZZ'])
>>> print C
XI |->  +XX
IX |->  +IX
ZI |->  +ZI
IZ |->  +ZZ
```

Also, the results of an element of the Clifford group can be left partially unspecified, using the singleton qecc.Unspecified:

```
>>> import qecc as q
>>> print q.Clifford(['IZ','XZ'],['XI',q.Unspecified])
XI |->  +IZ
IX |->  +XZ
ZI |->  +XI
IZ |-> Unspecified
```

Once an instance of *qecc.Clifford* has been constructed in this way, its action on elements of the Pauli group can be calculated by calling the Clifford instance as a function.

```
>>> from qecc import I, X, Y, Z
>>> C(X & Y)
i^0 YZ
>>> map(C, ['XI', 'IX', 'YI', 'IY', 'ZI', 'IZ'])
[i^0 XX, i^0 IX, i^0 YX, i^0 ZY, i^0 ZI, i^0 ZZ]
```

Note that in this example, C has converted strings to *qecc.Pauli* instances. This is done automatically by *qecc.Clifford*.

Instances of Clifford can be combined by multiplication (*) and by tensor products (&). Multiplication of two Clifford instances returns a new instance representing their composition, while the tensor product returns a new instance that acts on each register independently.

```
>>> import qecc as q
>>> C = q.Clifford(['XX', 'IX'], ['ZI', 'ZZ'])
>>> D = q.Clifford(['XI', 'IZ'], ['ZI', 'IX'])
>>> print C * D
XI |->  +XX
IX |->  +ZZ
```

(continues on next page)

(continued from previous page)

```
ZI |->  +ZI
IZ |->  +IX
>>> print C & D
X[0] |->  +X[0] X[1]
X[3] |->  +Z[3]
Z[1] |->  +Z[0] Z[1]
Z[3] |->  +X[3]
```

Note that in the second example, the printing of the Clifford operator has switched to a sparse format that suppresses printing lines for qubits that are not acted upon by the operator (in this case, qubits `1` and `2` are trivially acted upon by `C & D`).

As with *`qecc.Pauli`*, the length of a *`qecc.Clifford`* instance is defined as the number of qubits on which that instance acts. This information is also exposed as the property `nq`.

```
>>> import qecc as q
>>> C = q.Clifford(['XX', 'IX'], ['ZI', 'ZZ'])
>>> print len(C)
2
>>> print C.nq
2
```

Class Reference

class qecc.**Clifford**(*xbars*, *zbars*)

Class representing an element of the Cifford group on n qubits.

Parameters

- **xbars** (list of *`qecc.Pauli`* instances) – A list of operators $\bar{X}_i$ such that the represented Clifford operation C acts as $C(X_i) = \bar{X}_i$. Note that in order for the represented operator to be an automorphism, each $\bar{X}_i$ must have phase either 0 or 2. A warning will result if this condition is not met.
- **zbars** (list of *`qecc.Pauli`* instances) – See `xbars`.

nq

Returns the number of qubits on which this *`qecc.Clifford`* object acts.

n_unspecified

Returns the number of unspecifed outputs of this *`qecc.Clifford`* object.

str_sparse()

Provides a compact representation for *`qecc.Clifford`* objects, intended for use in the case where many of the outputs have small support.

is_valid(*quiet=True*)

Returns `True` if this instance represents a valid automorphism. In particular, this method returns `True` if all output phase assignments are either 0 or 2, and if all of the commutation relations on its outputs are obeyed. Unspecified outputs are ignored.

Parameters **quiet** (*`bool`*) – If set to `True`, this method will not print out any information, but will return `True` or `False` as described above. Otherwise, if the operator is not a valid Clifford operator, diagnostic information will be printed.

inv()

Calculates the inverse C^{-1} of this Clifford operator C, such that $C^{-1} \cdot C$ is the identity Clifford.

conjugate_pauli(*pauli*)

Given an instance of *qecc.Pauli* representing the operator P, calculates the mapping $CPC^\dagger$, where C is the operator represented by this instance.

Parameters **pauli** (qecc.Pauli) – Representation of the Pauli operator P.

Returns Representation of the Pauli operator $CPC^\dagger$, where C is the Clifford operator represented by this instance.

Return type *qecc.Pauli*

constraint_completions()

Yields an iterator onto possible Clifford operators whose outputs agree with this operator for all outputs that are specified. Note that all yielded operators assign the phase 0 to all outputs, by convention.

If this operator is fully specified, the iterator will yield exactly one element, which will be equal to this operator.

For example:

```
>>> import qecc as q
>>> C = q.Clifford([q.Pauli('XI'), q.Pauli('IX')], [q.Unspecified, q.
↪Unspecified])
>>> it = C.constraint_completions()
>>> print it.next()
XI |->  +XI
IX |->  +IX
ZI |->  +ZI
IZ |->  +IZ
>>> print it.next()
XI |->  +XI
IX |->  +IX
ZI |->  +ZI
IZ |->  +IY
>>> print len(list(C.constraint_completions()))
8
```

If this operator is not a valid Clifford operator, then this method will raise an *qecc.InvalidCliffordError* upon iteraton.

as_bsm()

Returns a representation of the Clifford operator as a binary symplectic matrix.

Return type *qecc.BinarySymplecticMatrix*

as_unitary()

Returns a `numpy.ndarray` containing a unitary matrix representation of this Clifford operator.

Raises a `RuntimeError` if NumPy cannot be imported.

circuit_decomposition(*include_pauli=True*)

Returns a *qecc.Circuit* object consisting of the circuit decomposition of *self.as_bsm()* and a *qecc.Pauli* object which ensures the output phases of the *qecc.Clifford* object are preserved.

Parameters **include_pauli** (*bool*) – If *True*, Pauli locations are added at the end of the returned circuit. If *False*, the returned *Circuit* is correct only up to a Pauli operator at the end.

Return type *Circuit*

Alternate Constructors

In addition to specifying the outputs of a Clifford operator acting on the elementary generators of the Pauli group, one can also create a `Clifford` instance by specifying the ouput of an operator on an arbitrary generating set. In particular, the function *`qecc.generic_clifford()`* takes the inputs and outputs of a given Clifford operator in order to create a *`qecc.Clifford`* instance.

`qecc.`**`generic_clifford`**`(`*paulis_in*, *paulis_out*`)`

Given two lists of *`qecc.Pauli`* instances, `paulis_in` and `paulis_out`, produces an instance `C` of *`qecc.Clifford`* such that `C(paulis_in[i]) == paulis_out[i]` for all `i` in `range(2 * nq)`, where `nq` is the length of each element of the two lists.

Each of `paulis_in` and `paulis_out` is assumed to be ordered such that the slice `[0:nq]` produces a list of logical X operators, and such that the slice `[nq:2*nq]` produces the logical Z operators.

Parameters

- **`paulis_in`** – A list of length `2 * nq` logical Pauli operators specifying the input constraints for the desired Clifford operation.
- **`paulis_out`** – A list of length `2 * nq` logical Pauli operators specifying the output constraints for the desired Clifford operation.

Returns A Clifford operator mapping the input constraints to the output constraints.

Return type *qecc.Clifford*

Iterators onto the Clifford Group

`qecc.`**`clifford_group`**`(`*nq*, *consider_phases=False*`)`

Given a number of qubits n, returns an iterator that produces all elements of $\mathcal{C}_n$, the Clifford group on n qubits.

Parameters

- **`nq`** (*`int`*) – The number of qubits upon which each yielded element will act.
- **`consider_phases`** (*`bool`*) – If `True`, then Clifford operators whose assignments of phases to the generators of the Pauli group differ will be treated as distinct. Otherwise, the yielded elements will be drawn from the group $\hat{\mathcal{C}}_n = \mathrm{Aut}(\hat{\mathcal{P}}_n / \{i^k I : k \in \mathbb{Z}_4\})$, such that the phases of the outputs are not considered.

Common Clifford Gates

The `qecc` package provides support for several common Clifford operators. These functions can be used to quickly analyze small circuits. For more extensive circuit support, please see *Circuit Manipulation and Simulation*.

`qecc.`**`eye_c`**`(`*nq*`)`

Yields the identity Clifford, defined to map every generator of the Pauli group to itself.

Return type *Clifford*

`qecc.`**`cnot`**`(`*nq*, *ctrl*, *targ*`)`

Yields the `nq`-qubit CNOT Clifford controlled on `ctrl`, acting a Pauli X on `targ`.

Return type *`qecc.Clifford`*

`qecc.`**`hadamard`**`(`*nq*, *q*`)`

Yields the `nq`-qubit Clifford, switching X and Z on qubit `q`, yielding a minus sign on Y.

Return type *`qecc.Clifford`*

`qecc.phase(nq, q)`

Yields the $\frac{\pi}{4}_z$-rotation Clifford, acting on qubit `q`.

Return type *qecc.Clifford*

`qecc.swap(nq, q1, q2)`

Yields the swap Clifford, on `nq` qubits, which swaps the Pauli generators on `q1` and `q2`.

Return type *qecc.Clifford*

`qecc.cz(nq, q1, q2)`

Yields the `nq`-qubit C-Z Clifford, acting on qubits `q1` and `q2`.

Return type *qecc.Clifford*

`qecc.pauli_gate(pauli)`

Imports an instance of the *qecc.Pauli* class into the *qecc.Clifford* class, representing a Pauli as a series of sign changes.

Return type *qecc.Clifford*

1.2 Collections of Pauli Operators

1.2.1 `qecc.PauliList`: Sequence type for Pauli operators

For convinenence, the `qecc` package provides a subclass of `list` intended for use with Pauli operators. `PauliList` instances can be created either by converting an existing instance of a sequence type, or by providing the elements of the new `PauliList`.

```
>>> import qecc as q
>>> L = ['I', 'X', 'Y', 'Z']
>>> print q.PauliList(L)
PauliList(i^0 I, i^0 X, i^0 Y, i^0 Z)
>>> print q.PauliList('XYZ', 'YZX', 'ZXY')
PauliList(i^0 XYZ, i^0 YZX, i^0 ZXY)
```

Tensor products of a `qecc.Pauli`` with a `PauliList` result in tensoring the given Pauli group element onto each element of the list.

```
>>> from qecc import X
>>> print q.PauliList(L) & X
PauliList(i^0 IX, i^0 XX, i^0 YX, i^0 ZX)
```

In general, a *qecc.PauliList* can be used anywhere that a list of *qecc.Pauli* instances is appropriate. For example, the constructor of *qecc.Clifford* accepts *qecc.PauliList* instances:

```
>>> import qecc as q
>>> C = q.Clifford(q.PauliList('XX', q.Unspecified), q.PauliList(q.Unspecified, q.
↪Pauli('ZZ', phase=2)))
>>> print C
XI |->  +XX
IX |-> Unspecified
ZI |-> Unspecified
IZ |->  -ZZ
```

Class Reference

class qecc.**PauliList**(**paulis*)

Subclass of `list` offering useful methods for lists of *qecc.Pauli* instances.

Parameters paulis – Instances either of `str` or *qecc.Pauli*, or the special object `qecc.Unspecified`. Strings are passed to the constructor of *qecc.Pauli* for convinenence.

pad(*extra_bits=0, lower_right=None*)

Takes a PauliList, and returns a new PauliList, appending `extra_bits` qubits, with stabilizer operators specified by `lower_right`.

Parameters

- **pauli_list_in** – list of Pauli operators to be padded.
- **extra_bits** (*int*) – Number of extra bits to be appended to the system.
- **lower_right** – list of *qecc.Pauli* operators, acting on *extra_bits* qubits.

Return type list of *qecc.Pauli* objects.

Example:

```
>>> import qecc as q
>>> pauli_list = q.PauliList('XXX', 'YIY', 'ZZI')
>>> pauli_list.pad(extra_bits=2, lower_right=q.PauliList('IX','ZI'))
PauliList(i^0 XXXII, i^0 YIYII, i^0 ZZIII, i^0 IIIIX, i^0 IIIZI)
```

generated_group(*coset_rep=None*)

Yields an iterator onto the group generated by this list of Pauli operators. See also *qecc.from_generators*.

stabilizer_subspace()

Returns a `numpy.ndarray` of shape `(n - k, 2 ** n)` containing an orthonormal basis for the mutual +1 eigenspace of each fully specified Pauli in this list. Here, `n` is taken to be the number of qubits and `k` is taken to be the number of independent Pauli operators in this list.

Raises a `RuntimeError` if NumPy cannot be imported.

For example, to find the Bell basis vector $|\beta_{00}\rangle$ using the stabilizer formalism:

```
>>> import qecc as q
>>> q.PauliList('XX', q.Unspecified, q.Unspecified, 'ZZ').stabilizer_
↪subspace()
array([[ 0.70710678+0.j,  0.00000000+0.j,  0.00000000+0.j,  0.70710678+0.j]])
```

Similarly, one can find the codewords of the phase-flip code $S = \langle XXI, IXX\rangle$:

```
>>> q.PauliList('XXI', 'IXX').stabilizer_subspace()
array([[ 0.50000000+0.j,  0.00000000-0.j,  0.00000000-0.j,  0.50000000+0.j,
         0.00000000-0.j,  0.50000000+0.j,  0.50000000+0.j,  0.00000000-0.j],
       [ 0.02229922+0.j,  0.49950250+0.j,  0.49950250+0.j,  0.02229922+0.j,
         0.49950250+0.j,  0.02229922+0.j,  0.02229922+0.j,  0.49950250+0.j]])
```

Note that in this second case, some numerical errors have occured; this method does not guarantee that the returned basis vectors are exact.

centralizer_gens(*group_gens=None*)

Returns the generators of the centralizer group $\mathrm{C}(P_1, \dots, P_k)$, where P_i is the i^{th} element of this list. See *qecc.Pauli.centralizer_gens()* for more information.

1.3 Binary Symplectic Form

1.3.1 Introduction

The `qecc` package provides support for elements of the Pauli and Clifford groups in binary symplectic form, including support for algorithms acting on these representations. Note that all classes and functions documented here depend on the `numpy` package. For more information on the binary symplectic representation, read *[CRSS96]*, Section 2.

1.3.2 `qecc.BinarySymplecticVector`: Binary symplectic representation of Pauli group elements

The class *`qecc.BinarySymplecticVector`* provides a means of representing elements of the Pauli group (neglecting global phases) using binary vectors a and b such that an element P of the Pauli group acting on n qubits is $X^a Z^b = X^{a_1} Z^{b_1} \otimes \ldots \otimes X^{a_n} Z^{b_n}$. Binary symplectic vectors can be obtained from a single binary list, two binary lists, or converted from another Pauli instance (removing the phase):

```
>>> import qecc as q
>>> a=[1, 0, 1]; b=[0, 1, 1]
>>> q.BinarySymplecticVector(a,b)==q.BinarySymplecticVector(a+b)
True
```

```
>>> import qecc as q
>>> a=[1, 0, 1]; b=[0, 1, 1]
>>> q.BinarySymplecticVector(a,b)
( 1 0 1 | 0 1 1 )
```

```
>>> import qecc as q
>>> q.Pauli('XYIYIIZ',2).as_bsv()
( 1 1 0 1 0 0 0 | 0 1 0 1 0 0 1 )
```

Class Reference

class qecc.**BinarySymplecticVector**(**args*)

Encapsulates a binary symplectic vector representing an element of the Pauli group on n qubits.

A new *`BinarySymplecticVector`* can be constructed using either a single NumPy array containing both the X and Z parts of the binary symplectic vector. Alternatively, a new vector can be instantiated using two NumPy arrays. For example, the following two invocations are equivalent:

```
>>> import qecc
>>> import numpy as np
>>> bsv = qecc.BinarySymplecticVector(np.array([1, 0, 0, 0, 0, 0]))
>>> bsv = qecc.BinarySymplecticVector(np.array([1, 0, 0]), np.array([0, 0, 0]))
```

The `len` of a *`BinarySymplecticVector`* is defined as the number of qubits upon which the represented Pauli operator acts, and is thus half of the length of a single array containing the same data.

x

Array containing the X part of the binary symplectic vector.

Return type `numpy.ndarray`, shape `(2 * nq, )`.

```
>>> import qecc as q
>>> q.BinarySymplecticVector([1,0,0,0,1,0]).x
array([1, 0, 0])
```

z

Array containing the Z part of the binary symplectic vector.

Return type `numpy.ndarray`, shape `(nq, )`.

```
>>> import qecc as q
>>> q.BinarySymplecticVector([1,0,0,0,1,0]).z
array([0, 1, 0])
```

copy()

Returns a copy of the binary symplectic vector such that mutations of the copy do not affect this instance. For more details, see the `numpy.ndarray.copy()` method.

as_pauli()

Returns an instance of `qecc.Pauli` representing the same Pauli operator as this vector. Note that phase information is not preserved by the binary symplectic representation of the Pauli group, and so `P.as_bsv().as_pauli()` need not equal `P`.

```
>>> import qecc as q
>>> pauli_with_phase=q.Pauli('IXXYZ',2)
>>> pauli_with_phase.as_bsv().as_pauli()
i^0 IXXYZ
```

bsip(*other*)

Returns the binary symplectic inner product $u \odot v$ of this vector with another vector. Letting $u = (a|b)$ and $v = (c|d)$, $u \odot v = a \cdot d + b \cdot c$.

```
>>> import qecc as q
>>> vector_a = q.BinarySymplecticVector([1,0,1],[0,1,1])
>>> vector_b = q.Pauli('YYZ').as_bsv()
>>> vector_a.bsip(vector_b)
1
```

Utility Functions

`qecc.`**all_pauli_bsvs**(*nq*)

Lists all the Paulis on `nq` qubits according to their binary symplectic representations.

Parameters **nq** (`int`) – Number of qubits.

Returns an iterator that yields the binary symplectic representations of each element of the Pauli group $\mathcal{P}_n$.

```
>>> list(all_pauli_bsvs(1))
[( 0 | 0 ), ( 0 | 1 ), ( 1 | 0 ), ( 1 | 1 )]
```

`qecc.`**constrained_set**(*pauli_array_input*, *logical_array_input*)

Given a set of constraints of the form $P_i \odot Q = b_i$, with each P_i a Pauli operator and each b_i a bit, yields an iterator onto Pauli operators Q such that all constraints are satisfied.

Parameters

- **pauli_array_input** (`list` of `qecc.Pauli` instances.) – Constraint operators P_i.

- **logical_array_input** (`numpy.ndarray` of *dtype=int* and shape `(len(pauli_array_input), )`.) – Constraint values b_i.

```
>>> import qecc as q
>>> list(q.constrained_set(map(lambda s: q.Pauli(s).as_bsv(), ['XY','ZZ']),[1,0]))
[( 0 0 | 0 1 ), ( 0 0 | 1 0 ), ( 1 1 | 0 0 ), ( 1 1 | 1 1 )]
```

`qecc.`**`commute`**`(`*bsv1*, *bsv2*`)`

Returns True if bsv1 and bsv2 commute by evaluating the symplectic inner product.

Return type bool

`qecc.`**`xz_switch`**`(`*bsv*`)`

Given a `qecc.BinarySymplecticVector`, returns a new vector whose X and Z parts have been swapped.

1.3.3 `qecc.BinarySymplecticMatrix` - Binary symplectic representation of Clifford group elements

Class Reference

class `qecc.`**`BinarySymplecticMatrix`**`(`**args*`)`

Encapsulates a binary symplectic matrix representing an element of the Clifford group on n qubits.

A new `BinarySymplecticMatrix` can be constructed using either a single NumPy 2-D array containing the XX, XZ, ZX, and ZZ parts of the binary symplectic matrix. Alternatively, a new matrix can be instantiated using four NumPy arrays. For example, the following two invocations are equivalent:

```
>>> import qecc
>>> import numpy as np
>>> bsm = qecc.BinarySymplecticMatrix(np.array([[1, 0, 0, 0],[1, 1, 0, 0],[0, 0,␣
↪1, 1],[0, 0, 0, 1]]))
>>> bsm = qecc.BinarySymplecticMatrix(np.array([[1, 0],[1, 1]]), np.array([[0, 0],
↪[0, 0]]), np.array([[0, 0],[0, 0]]), np.array([[1, 1],[0, 1]]))
```

nq

Returns the number of qubits that the binary symplectic matrix acts upon.

xc

Returns the left half of a binary symplectic matrix.

zc

Returns the right half of a binary symplectic matrix.

xr

Returns the top half of a binary symplectic matrix.

zr

Returns the bottom half of a binary symplectic matrix.

xx

Returns the upper-left quadrant of a binary symplectic matrix.

xz

Returns the upper-right quadrant of a binary symplectic matrix.

zx

Returns the lower-left quadrant of a binary symplectic matrix.

zz
Returns the lower-right quadrant of a binary symplectic matrix.

left_H (*j*)
Multiplies on the left by a Hadamard gate on the j^{th} qubit. This method acts in-place, as opposed to acting on a copy of the binary symplectic matrix. In order to preserve the original matrix, use the `copy()` method:

```
>>> new_bsm = bsm.copy().left_H(idx)
```

right_H (*j*)
Multiplies on the right by a Hadamard gate on the j^{th} qubit. See `left_H()` for more details.

right_H_all ()
Multiplies on the right by a Hadamard gate on each qubit. See `left_H()` for more details.

left_SWAP (*j*, *k*)
Multiplies on the left by a SWAP gate between the j^{th} and k^{th} qubits. This method acts in-place, as opposed to acting on a copy of the binary symplectic matrix. In order to preserve the original matrix, use the `copy()` method:

```
>>> new_bsm = bsm.copy().left_SWAP(j, k)
```

right_SWAP (*j*, *k*)
Multiplies on the right by a SWAP gate between the j^{th} and k^{th} qubits. See `left_SWAP()` for more details.

left_CNOT (*c*, *t*)
Multiplies on the left by a CNOT gate controlled by the c^{th} qubit and targeting the k^{th} qubit. This method acts in-place, as opposed to acting on a copy of the binary symplectic matrix. In order to preserve the original matrix, use the `copy()` method:

```
>>> new_bsm = bsm.copy().left_CNOT(c, t)
```

right_CNOT (*c*, *t*)
Multiplies on the right by a CNOT gate controlled by the c^{th} qubit and targeting the k^{th} qubit. For more details, see `left_CNOT()`.

left_R_pi4 (*i*)
Multiplies on the left by an $R_{\pi/4}$ gate acting on the i^{th} qubit. This method acts in-place, as opposed to acting on a copy of the binary symplectic matrix. In order to preserve the original matrix, use the `copy()` method:

```
>>> new_bsm = bsm.copy().left_R_pi4(c, t)
```

right_R_pi4 (*i*)
Multiplies on the right by an $R_{\pi/4}$ gate acting on the i^{th} qubit. For more details, see `left_R_pi4()`.

left_CZ (*c1*, *c2*)
Multiplies on the left by an controlled-Z gate acting between the c_1^{th} and c_2^{th} qubits. This method acts in-place, as opposed to acting on a copy of the binary symplectic matrix. In order to preserve the original matrix, use the `copy()` method:

```
>>> new_bsm = bsm.copy().left_CZ(c, t)
```

right_CZ (*c1*, *c2*)
Multiplies on the right by an controlled-Z gate acting between the c_1^{th} and c_2^{th} qubits. For more details, see `left_CZ()`.

inv(*check_validity=True*)

Returns the inverse of this binary symplectic matrix, assuming that this matrix represents a valid Clifford gate.

Note that if the matrix H does not represent a valid Clifford, this method will return a matrix G such that HG is not the identity matrix.

Parameters **check_validity** (*bool*) – If `True`, then the matrix is first checked to ensure that it is a valid Clifford.

Raises *qecc.InvalidCliffordError* if `check_validity` is `True` and the binary symplectic matrix being inverted does not represent a valid Clifford group element.

as_clifford(*check_validity=True*)

Converts this binary symplectic matrix into a Clifford representation.

Parameters **check_validity** (*bool*) – If `True`, then the matrix is first checked to ensure that it is a valid Clifford.

Return type *qecc.Clifford*

Returns The same gate as this binary symplectic matrix, represented as an instance of *qecc.Clifford*.

is_valid()

Checks the satisfaction of the symplectic condition on a *qecc.BinarySymplecticMatrix* object.

copy()

Returns a copy of this binary symplectic matrix, pointing to a distinct location in memory.

circuit_decomposition(*validate=True*)

Decomposes the binary symplectic matrix using the algorithm of *[AG04]*.

Utility Functions

qecc.**is_bsm_valid**(**args*, ***kwargs*)

qecc.**bsmzeros**(*nq*)

Returns a binary symplectic matrix on n qubits, initialized to all zeros.

Parameters **nq** (*int*) – Number of qubits that the created matrix will act upon.

Returns A binary symplectic matrix containing all zeros.

Return type *BinarySymplecticMatrix*

qecc.**array_to_pauli**(*bsv_array*)

Function wrapper for type conversion from binary symplectic vector to *qecc.Pauli*. See *qecc.BinarySymplecticVector.as_pauli()*.

1.4 Stabilizer Codes

1.4.1 `qecc.StabilizerCode`

Introduction

QuaEC includes a class, *qecc.StabilizerCode*, that represents error-correcting codes specified using the stabilizer formalism *[Got97]*. To construct a stabilizer code in QuaEC, the generators of a stabilizer group must be specified along with a particular assignment of logical operators acting on states encoded in the stabilizer code.

```
>>> import qecc as q
>>> stab = q.StabilizerCode(['ZZI', 'IZZ'], ['XXX'], ['ZZZ'])
>>> print stab
S = <i^0 ZZI, i^0 IZZ>
Xbars = PauliList(i^0 XXX)
Zbars = PauliList(i^0 ZZZ)
```

For convienence, several static methods are provided to create instances for well-known stabilizer codes.

```
>>> stab = q.StabilizerCode.perfect_5q_code()
>>> print stab
5-qubit perfect code
S = <i^0 XZZXI, i^0 IXZZX, i^0 XIXZZ, i^0 ZXIXZ>
Xbars = PauliList(i^0 XXXXX)
Zbars = PauliList(i^0 ZZZZZ)
```

Once constructed, an instance of *qecc.StabilizerCode* exposes properties that describe the number of physical and logical qubits, as well as the distance of the code. (Please note that calculating the distance can be extremely slow for large codes.)

```
>>> print (stab.nq, stab.nq_logical, stab.distance)
(5, 1, 3)
```

Encoders and decoders for stabilizer codes can be found in a straightforward manner using *qecc.StabilizerCode*.

```
>>> enc = stab.encoding_cliffords().next()
>>> print enc
X[0] |->  +X[0] X[1] X[2] X[3] X[4]
X[1] |->  +X[0] X[2] X[3] X[4]
X[2] |->  +X[1] X[2]
X[3] |->  +Y[0] X[1] X[3] Y[4]
X[4] |->  +X[0] X[1] Y[3] Y[4]
Z[0] |->  +Z[0] Z[1] Z[2] Z[3] Z[4]
Z[1] |->  +X[0] Z[1] Z[2] X[3]
Z[2] |->  +X[1] Z[2] Z[3] X[4]
Z[3] |->  +X[0] X[2] Z[3] Z[4]
Z[4] |->  +Z[0] X[1] X[3] Z[4]
>>> print enc.inv()
X[0] |->  -X[0] Z[3] X[4]
X[1] |->  +X[0] X[1]
X[2] |->  +X[0] X[1] X[2]
X[3] |->  -X[0] X[2] X[3] Z[4]
X[4] |->  +X[0] Y[3] Y[4]
Z[0] |->  -Z[0] Y[1] Y[3] Z[4]
Z[1] |->  -Z[0] Y[2] Z[3] Y[4]
Z[2] |->  +Z[0] Z[1] Z[2] X[3]
Z[3] |->  -Z[0] Y[1] Z[3] Y[4]
Z[4] |->  -Z[0] Z[1] X[2] Z[3] Z[4]
```

Stabilizer codes may be combined by the tensor product (reprsented in QuaEC by `&`), or by concatenation:

```
>>> print stab & stab
S = <i^0 XZZXIIIIII, i^0 IXZZXIIIII, i^0 XIXZZIIIII, i^0 ZXIXZIIIII, i^0 IIIIIXZZXI,␣
↪i^0 IIIIIIXZZX, i^0 IIIIIXIXZZ, i^0 IIIIIZXIXZ>
Xbars = PauliList(i^0 XXXXXIIIII, i^0 IIIIIXXXXX)
Zbars = PauliList(i^0 ZZZZZIIIII, i^0 IIIIIZZZZZ)
```

(continues on next page)

(continued from previous page)

```
>>> print q.StabilizerCode.bit_flip_code(1).concatenate(q.StabilizerCode.phase_flip_
↪code(1))
S = <i^0 Z[0] Z[1], i^0 Z[1] Z[2], i^0 Z[3] Z[4], i^0 Z[4] Z[5], i^0 Z[6] Z[7], i^0␣
↪Z[7] Z[8], i^0 XXXXXXIII, i^0 IIIXXXXXX>
Xbars = PauliList(i^0 XXXXXXXXX)
Zbars = PauliList(i^0 ZZZZZZZZZ)
```

Class Reference

class qecc.**StabilizerCode**(*group_generators*, *logical_xs*, *logical_zs*, *label=None*)

Class representing a stabilizer code specified by the generators of its stabilizer group and by representatives for the logical operators acting on the code.

Parameters

- **group_generators** – Generators N_i such that the stabilizer group S of the represented code is given by $S = \langle N_i \rangle$.
- **logical_xs** – Representatives for the logical X operators acting on encoded states.
- **logical_zs** – Representatives for the logical Z operators acting on encoded states.
- **label** (*str*) – User-facing name for the stabilizer code.

nq

The number of physical qubits into which this code encodes.

n_constraints

The number of stabilizer constraints on valid codewords.

nq_logical

The number of logical qubits admitted by this code.

logical_ys

Derives logical Y operators, given logical X and Z operators.

logical_ops

Returns a list of all logical operators for a code in the form [Xs, Ys, Zs].

distance

The distance of this code, defined by $\min \mathrm{wt}\{P | P \in \mathrm{N}(S) \backslash S\}$, where S is the stabilizer group for this code.

Warning: this property is currently very slow to compute.

n_correctable

The number of errors t correctable by this code, defined by $\left\lfloor \frac{d-1}{2} \right\rfloor$, where d is the distance of the code, given by the `distance` property.

stabilizer_group(*coset_rep=None*)

Iterator onto all elements of the stabilizer group S describing this code, or onto a coset PS of the stabilizer group.

Parameters **coset_rep** (`qecc.Pauli`) – A Pauli operator P, so that the iterated coset is PS. If not specified, defaults to the identity.

Yields All elements of the coset PS of the stabilizer group S.

logical_pauli_group(*incl_identity=True*)

Iterator onto the group $\mathrm{N}(S)/S$, where S is the stabilizer group describing this code. Each member of the group is specified by a coset representative drawn from the respective elements of $\mathrm{N}(S)/S$. These representatives are chosen to be the logical X and Z operators specified as properties of this instance.

Parameters **`incl_identity`** (*`bool`*) – If `False`, the identity coset S is excluded from this iterator.

Yields A representative for each element of $\mathrm{N}(S)/S$.

`normalizer_group`(*mod_s=False*)

Returns all elements of the normalizer of the stabilizer group. If `mod_s` is `True`, returns the set $N(S)\backslash S$.

`encoding_cliffords`()

Returns an iterator onto all Clifford operators that encode into this stabilizer code, starting from an input register such that the state to be encoded is a state of the first k qubits, and such that the rest of the qubits in the input register are initialized to $|0\rangle$.

Yields instances `C` of *`qecc.Clifford`* such that `C(q.StabilizerCode.unencoded_state(k, n - k))` equals this code.

`syndrome_to_recovery_operator`(*synd*)

Returns a Pauli operator which corrects an error on the stabilizer code `self`, given the syndrome `synd`, a bitstring indicating which generators the implied error commutes with and anti-commutes with.

Parameters **`synd`** – a string, list, tuple or other sequence type with entries consisting only of 0 or 1. This parameter will be certified before use.

`syndromes_and_recovery_operators`()

Outputs an iterator onto tuples of syndromes and appropriate recovery operators.

`recovery_circuit_as_qcircuit`(*C=None*, *R=None*)

Returns the recovery operator (as specified by *`syndromes_and_recovery_operators()`*), expressed as a Qcircuit array.

Parameters

- **`C`** (*`float`*) – Width (in ems) of each column.
- **`R`** (*`float`*) – Height (in ems) of each column.

`star_decoder`(*for_enc=None*, *as_dict=False*)

Returns a tuple of a decoding Clifford and a *`qecc.PauliList`* specifying the recovery operation to perform as a function of the result of a $Z^{\otimes n-k}$ measurement on the ancilla register.

For syndromes corresponding to errors of weight greater than the distance, the relevant element of the recovery list will be set to `qecc.Unspecified`.

Parameters

- **`for_enc`** – If not `None`, specifies to use a given Clifford operator as the encoder, instead of the first element yielded by *`encoding_cliffords()`*.
- **`as_dict`** (*`bool`*) – If `True`, returns a dictionary from recovery operators to syndromes that indicate that recovery.

`minimize_distance_from`(*other*, *quiet=True*)

Reorders the stabilizer group generators of this code to minimize the Hamming distance with the group generators of another code, using a greedy heuristic algorithm.

`stabilizer_subspace`()

Returns a $2^k \times 2^n$ array whose rows form a basis for the codespace of this code. Please note that by necessity, this code is exponentially slow as a function of the numbers of physical and logical qubits.

block_logical_pauli(*P*)
Given a Pauli operator P acting on k, finds a Pauli operator $\overline{P}$ on n_k qubits that corresponds to the logical operator acting across k blocks of this code.

Note that this method is only supported for single logical qubit codes.

measure_gen_onto_ancilla(*gen_idx*)
Produces a circuit that measures the stabilizer code generator `self.group_generators[gen_idx]` onto the qubit labelled by `stab.nq` (that is, the next qubit not in the physical register used by the code).

Parameters gen_idx (*int*) – Index of a generator of the stabilizer group, as specified by the `group_generators` property of this instance.

Returns qecc.Circuit A circuit that maps a measurement of `group_generators[gen_idx]` onto a measurement of Z on the ancilla qubit alone.

syndrome_meas_circuit()
Returns a circuit which measures all stabilizer generators onto ancillae, using `measure_gen_onto_ancilla`.

permute_gen_ops(*perm*)
Returns a stabilizer code with generators related to the generators of *self*, with every instance of {X,Y,Z} replaced with {perm[0],perm[1],perm[2]}.

Parameters perm (*list*) – A list containing 'X','Y', and 'Z' in any order, indicating which permutation is to be applied.

```
>>> new_stab = StabilizerCode.bit_flip_code(1).permute_gen_ops('ZYX')
>>> assert new_stab.group_generators == StabilizerCode.phase_flip_code(1).
↪group_generators
```

concatenate(*other*)
Returns the stabilizer for a concatenated code, given the stabilizers for two codes. At this point, it only works for two $k = 1$ codes.

transcoding_cliffords(*other*)
Returns an iterator onto all *qecc.Clifford* objects which take states specified by `self`, and return states specified by `other`.

Parameters other – *qecc.StabilizerCode*

min_len_transcoding_clifford(*other*)
Searches the iterator provided by *transcoding_cliffords* for the shortest circuit decomposition.

static ancilla_register(*nq=1*)
Creates an instance of *qecc.StabilizerCode* representing an ancilla register of `nq` qubits, initialized in the state $|0\rangle^{\otimes \mathrm{nq}}$.

Return type *qecc.StabilizerCode*

static unencoded_state(*nq_logical=1*, *nq_ancilla=0*)
Creates an instance of *qecc.StabilizerCode* representing an unencoded register of `nq_logical` qubits tensored with an ancilla register of `nq_ancilla` qubits.

Parameters nq_logical (*int*) – Number of qubits to

Return type *qecc.StabilizerCode*

static flip_code(*n_correctable*, *stab_kind='Z'*)
Creates an instance of *qecc.StabilizerCode* representing a code that protects against weight-`n_correctable` flip errors of a single kind.

This method generalizes the bit-flip and phase-flip codes, corresponding to `stab_kind=qecc.Z` and `stab_kind=qecc.X`, respectively.

Parameters

- **n_correctable** (*int*) – Maximum weight of the errors that can be corrected by this code.
- **stab_kind** (`qecc.Pauli`) – Single-qubit Pauli operator specifying which kind of operators to use for the new stabilizer code.

Return type *qecc.StabilizerCode*

static bit_flip_code(*n_correctable*)

Creates an instance of *qecc.StabilizerCode* representing a code that protects against weight-`n_correctable` bit-flip errors.

Parameters **n_correctable** (*int*) – Maximum weight of the bit-flip errors that can be corrected by this code.

Return type *qecc.StabilizerCode*

static phase_flip_code(*n_correctable*)

Creates an instance of *qecc.StabilizerCode* representing a code that protects against weight-`n_correctable` phase-flip errors.

Parameters **n_correctable** (*int*) – Maximum weight of the phase-flip errors that can be corrected by this code.

Return type *qecc.StabilizerCode*

static perfect_5q_code()

Creates an instance of *qecc.StabilizerCode* representing the 5-qubit perfect code.

Return type *qecc.StabilizerCode*

static steane_code()

Creates an instance of *qecc.StabilizerCode* representing the 7-qubit Steane code.

Return type *qecc.StabilizerCode*

static shor_code()

Creates an instance of *qecc.StabilizerCode* representing the 9-qubit Shor code.

Return type *qecc.StabilizerCode*

static css_code(*C1*, *C2*)

Not yet implemented.

static reed_muller_code(*r*, *t*)

Not yet implemented.

static reed_solomon_code(*r*, *t*)

Not yet implemented.

1.5 Circuit Manipulation and Simulation

1.5.1 Introduction

Quantum circuits are modeled in QuaEC by a sequence type, *qecc.Circuit*, that stores zero or more circuit elements, known as *locations*. Each location has a *kind* that indicates if it is a gate, measurement or preparation location, as well as which gate, which measurement or which preparation is indicated.

Creating a `qecc.Location` instance consists of specifying the kind of location along with a sequence of indices indicating which qubits that location acts upon.

```
>>> import qecc as q
>>> loc = q.Location('CNOT', 0, 2)
```

The `qecc.Location.as_clifford()` method allows converting gate locations back into a `qecc.Clifford` representation if applicable.

```
>>> print loc.as_clifford()
XII |->  +XIX
IXI |->  +IXI
IIX |->  +IIX
ZII |->  +ZII
IZI |->  +IZI
IIZ |->  +ZIZ
```

When creating a `qecc.Circuit`, you may specify each location either as an instance of `qecc.Location` or as a tuple of arguments to `qecc.Location`'s constructor.

```
>>> circ = q.Circuit(('CNOT', 0, 2), ('H', 1), ('X', 0))
```

Printing a circuit or location results in that instance being represented in the QuASM format, a plaintext representation of quantum circuits.

```
>>> print loc
    CNOT    0 2
>>> print circ
    CNOT    0 2
    H       1
    X       0
```

The number of qubits, depth and size of each location and circuit can be found by querying the appropriate properties of a `qecc.Location` or `qecc.Circuit`:

```
>>> print loc.nq
3
>>> print circ.nq, circ.depth, circ.size, len(circ)
3 2 3 3
```

Once constructed, a `qecc.Circuit` can be transformed in several ways, including simplifications and representations in terms of depth-1 subcircuits.

```
>>> circ = q.Circuit(('CNOT', 0, 2), ('H', 1), ('X', 0), ('H', 1))
>>> print circ
    CNOT    0 2
    H       1
    X       0
    H       1
>>> print circ.cancel_selfinv_gates()
    CNOT    0 2
    X       0
>>> circ = q.Circuit(('CZ', 0, 2), ('H', 1), ('X', 0))
>>> print circ.replace_cz_by_cnot()
    H       2
    CNOT    0 2
```

(continues on next page)

(continued from previous page)

```
    H       2
    H       1
    X       0
>>> print "\n    --\n".join(map(str, circ.group_by_time()))
    H       2
    --
    CNOT    0 2
    --
    H       2
    H       1
    X       0
```

Note that, except for *qecc.Circuit.group_by_time()*, each of these transformations mutates the circuit, so that the original circuit is lost.

```
>>> print circ
    H       2
    CNOT    0 2
    H       2
    H       1
    X       0
```

If a circuit consists entirely of Clifford gate locations, then its entire action may be represented as a *qecc.Clifford* instance:

```
>>> circ = q.Circuit(('CZ', 0, 2), ('H', 1), ('X', 0))
>>> print circ.as_clifford()
XII |->  +XIZ
IXI |->  +IZI
IIX |->  -ZIX
ZII |->  -ZII
IZI |->  +IXI
IIZ |->  +IIZ
```

Finally, circuits can be exported to QCViewer files (`*.qcv`) for easy integration with QCViewer's functionality.

```
>>> print circ.as_qcviewer()
.v q1 q2 q3
.i q1
.o q1
BEGIN
    Z    q1 q3
    H    q2
    X    q1
END
```

Note that, by default, qubits in the QCViewer export are named "q1", "q2" and so on. This may be overriden by passing a sequence of strings as the `qubit_names` argument. Which qubits get assigned to the `.i` and `.o` headers in the QCViewer file are controlled by the `inputs` and `outputs` arguments, respectively.

```
>>> print circ.as_qcviewer(inputs=(0,), outputs=(0,), qubit_names=["in1", "anc1",
↪"anc2"])
.v in1 anc1 anc2
.i in1
.o in1
BEGIN
```

(continues on next page)

(continued from previous page)

```
    Z    in1 anc2
    H    anc1
    X    in1
END
```

1.5.2 `qecc.Location`: Class representing locations in a circuit

Class Reference

class qecc.**Location**(*kind*, **qubits*)

Represents a gate, wait, measurement or preparation location in a circuit.

Note that currently, only gate locations are implemented.

Parameters

- **kind** (*int or str*) – The kind of location to be created. Each kind is an abbreviation drawn from `Location.KIND_NAMES`, or is the index in `Location.KIND_NAMES` corresponding to the desired location kind.
- **qubits** (*tuple of ints.*) – Indicies of the qubits on which this location acts.

KIND_NAMES = ['I', 'X', 'Y', 'Z', 'H', 'R_pi4', 'CNOT', 'CZ', 'SWAP']

Names of the kinds of locations used by QuaEC.

static from_quasm(*source*)

Returns a *qecc.Location* initialized from a QuASM-formatted line.

Return type *qecc.Location*

Returns The location represented by the given QuASM source.

kind

Returns a string defining which kind of location this instance represents. Guaranteed to be a string that is an element of `Location.KIND_NAMES`.

qubits

Returns a tuple of ints describing which qubits this location acts upon.

nq

Returns the number of qubits in the smallest circuit that can contain this location without relabeling qubits. For a *qecc.Location* `loc`, this property is defined as `1 + max(loc.nq)`.

is_clifford

Returns `True` if and only if this location represents a gate drawn from the Clifford group.

wt

Returns the number of qubits on which this location acts.

as_clifford(*nq=None*)

If this location represents a Clifford gate, returns the action of that gate. Otherwise, a `RuntimeError` is raised.

Parameters **nq** (*int*) – Specifies how many qubits to represent this location as acting upon. If not specified, defaults to the value of the `nq` property.

Return type *qecc.Clifford*

as_qcviewer(*qubit_names=None*)

Returns a representation of this location in a format suitable for inclusion in a QCViewer file.

Parameters **qubit_names** – If specified, the given aliases will be used for the qubits involved in this location when exporting to QCViewer. Defaults to "q1", "q2", etc.

Return type str

Note that the identity (or "wait") location requires the following to be added to QCViewer's `gateLib`:

```
NAME wait
DRAWNAME "1"
SYMBOL I
1 , 0
0 , 1
```

relabel_qubits(*relabel_dict*)

Returns a new location related to this one by a relabeling of the qubits. The relabelings are to be indicated by a dictionary that specifies what each qubit index is to be mapped to.

```
>>> import qecc as q
>>> loc = q.Location('CNOT', 0, 1)
>>> print loc
    CNOT    0 1
>>> print loc.relabel_qubits({1: 2})
    CNOT    0 2
```

Parameters **relabel_dict** (`dict`) – If *i* is a key of *relabel_dict*, then qubit *i* will be replaced by *relabel_dict[i]* in the returned location.

Return type `qecc.Location`

Returns A new location with the qubits relabeled as specified by *relabel_dict*.

1.5.3 `qecc.Circuit`: Class modeling arrangements of locations

Class Reference

class qecc.**Circuit**(**locs*)

append(*newval*)

L.append(object) – append object to end

insert(*at*, *newval*)

L.insert(index, object) – insert object before index

nq

Returns the number of qubits on which this circuit acts.

size

Returns the number of locations in this circuit. Note that this property is synonymous with `len`, in that `len(circ) == circ.size` for all `qecc.Circuit` instances.

depth

Returns the minimum number of timesteps required to implement exactly this circuit in parallel.

static from_quasm(*source*)

Returns a `qecc.Circuit` object from a QuASM-formatted file, producing one location per line.

as_quasm()

Returns a representation of the circuit in an assmembler-like format. In this format, each location is represented by a single line where the first field indicates the kind of location and the remaining fields indicate the qubits upon which the location acts.

```
>>> import qecc as q
>>> circ = q.Circuit(('CNOT', 0, 2), ('H', 2), ('SWAP', 1, 2), ('I', 0))
>>> print circ.as_quasm()
    CNOT    0 2
    H       2
    SWAP    1 2
    I       0
```

as_qcviewer(*inputs=(0,), outputs=(0,), qubit_names=None*)
Returns a string representing this circuit in the format recognized by QCViewer.

Parameters

- **inputs** (*tuple*) – Specifies which qubits should be marked as inputs in the exported QCViewer circuit.
- **outputs** (*tuple*) – Specifies which qubits should be marked as outputs in the exported QCViewer circuit.
- **qubit_names** – Names to be used for each qubit when exporting to QCViewer.

as_qcircuit(*C=None, R=None*)
Typesets this circuit using the Qcircuit package for $\LaTeX$.

Parameters

- **C** (*float*) – Width (in ems) of each column.
- **R** (*float*) – Height (in ems) of each column.

Return type `str`

Returns A string containing $\LaTeX$ source code for use with Qcircuit.

as_clifford()
If this circuit is composed entirely of Clifford operators, converts it to a *qecc.Clifford* instance representing the action of the entire circuit. If the circuit is not entirely Clifford gates, this method raises a `RuntimeError`.

cancel_selfinv_gates(*start_at=0*)
Transforms the circuit, removing any self-inverse gates from the circuit if possible. Note that not all self-inverse gates are currently supported by this method.

Parameters **start_at** (*int*) – Specifies which location to consider first. Any locations before `start_at` are not considered for cancelation by this method.

replace_cz_by_cnot()
Changes all controlled-Z gates in this circuit to controlled-NOT gates, adding Hadamard locations as required.

group_by_time(*pad_with_waits=False*)
Returns an iterator onto subcircuits of this circuit, each of depth 1.

Parameters **pad_with_waits** (*bool*) – If `True`, each subcircuit will have wait locations added such that every qubit is acted upon in every subcircuit.

Yields each depth-1 subcircuit, corresponding to time steps of the circuit

pad_with_waits()
Returns a copy of the *qecc.Circuit* `self`, which contains explicit wait locations.

relabel_qubits(*relabel_dict*)
Returns a new circuit related to this one by a relabeling of the qubits. The relabelings are to be indicated by a dictionary that specifies what each qubit index is to be mapped to.

```
>>> import qecc as q
>>> loc = q.Location('CNOT', 0, 1)
>>> print loc
    CNOT    0 1
>>> print loc.relabel_qubits({1: 2})
    CNOT    0 2
```

Parameters **relabel_dict** (*dict*) – If *i* is a key of *relabel_dict*, then qubit *i* will be replaced by *relabel_dict[i]* in the returned circuit.

Return type *qecc.Circuit*

Returns A new circuit with the qubits relabeled as specified by *relabel_dict*.

Functions Acting on `qecc.Circuit`

qecc.**propagate_fault**(*circuitlist*, *fault*)
Given a list of circuits representing a list of timesteps (see *qecc.Circuit.group_by_time()*) and a Pauli fault, propagates that fault through the remainder of the time-sliced circuit.

Parameters

- **circuitlist** (*list*) – A list of *qecc.Circuit* instances representing the timesteps of a larger circuit.
- **fault** (qecc.Pauli) – A Pauli fault to occur immediately before timestep `timestep`.
- **timestep** (*int*) – The timestep immediately following when the fault to be propagated occured.

Return type *qecc.Pauli*

Returns The effective fault after propagating `fault` through the remainder of `circuitlist`.

qecc.**possible_faults**(*circuit*)
Takes a sub-circuit which has been padded with waits, and returns an iterator onto Paulis which may occur as faults after this sub-circuit.

Parameters **circuit** (qecc.Circuit) – Subcircuit to in which faults are to be considered.

qecc.**possible_output_faults**(*circuitlist*)
Gives an iterator onto all possible effective faults due to 1-fault paths occuring within `circuitlist`, assuming it has been padded with waits.

Parameters **circuitlist** (*list*) – A list of *qecc.Circuit* instances representing timesteps in a larger circuit. See *qecc.Circuit.group_by_time()*.

Yields *qecc.Pauli* instances representing possible effective faults due to 1-fault paths within the circuit represented by `circuitlist`.

1.6 Constraint Solvers

1.6.1 Commutation Constraints

qecc.**solve_commutation_constraints**(*commutation_constraints=[]*, *anticommutation_constraints=[]*, *search_in_gens=None*, *search_in_set=None*)

Given commutation constraints on a Pauli operator, yields an iterator onto all solutions of those constraints.

Parameters

- **commutation_constraints** – A list of operators $\{A_i\}$ such that each solution P yielded by this function must satisfy $[A_i, P] = 0$ for all i.
- **anticommutation_constraints** – A list of operators $\{B_i\}$ such that each solution P yielded by this function must satisfy $\{B_i, P\} = 0$ for all i.
- **search_in_gens** – A list of operators $\{N_i\}$ that generate the group in which to search for solutions. If `None`, defaults to the elementary generators of the pc.Pauli group on n qubits, where n is given by the length of the commutation and anticommutation constraints.
- **search_in_set** – An iterable of operators to which the search for satisfying assignments is restricted. This differs from `search_in_gens` in that it specifies the entire set, not a generating set. When this parameter is specified, a brute-force search is executed. Use only when the search set is small, and cannot be expressed using its generating set.

Returns An iterator `it` such that `list(it)` contains all operators within the group $G = \langle N_1, \ldots, N_k \rangle$ given by `search_in_gens`, consistent with the commutation and anticommutation constraints.

This function is based on finding the generators of the centralizer groups of each commutation constraint, and is thus faster than a predicate-based search over the entire group of interest. The resulting iterator can be used in conjunction with other filters, however.

```
>>> import qecc as q
>>> list(q.solve_commutation_constraints(q.PauliList('XXI', 'IZZ', 'IYI'), q.
→PauliList('YIY')))
[i^0 XII, i^0 IIZ, i^0 YYX, i^0 ZYY]
>>> from itertools import ifilter
>>> list(ifilter(lambda P: P.wt <= 2, q.solve_commutation_constraints(q.PauliList(
→'XXI', 'IZZ', 'IYI'), q.PauliList('YIY'))))
[i^0 XII, i^0 IIZ]
```

1.7 Predicates and Filters

1.7.1 `qecc.Predicate`: Class representing predicate functions

The `qecc` package provides a class `Predicate` to represent a predicate function; that is, a function which returns a `bool`.

class qecc.**Predicate**(*fn*)

Class representing a predicate function on one or more arguments.

```
>>> from qecc import Predicate
>>> p = Predicate(lambda x: x > 0)
```

(continues on next page)

(continued from previous page)

```
>>> p(1)
True
>>> p(-1)
False
```

Instances can also be constructed by logical operations on existing Predicate instances:

```
>>> q = Predicate(lambda x: x < 3)
>>> (p & q)(1)
True
>>> (p | q)(-1)
True
>>> (~p)(2)
False
```

combine(*other*, *outer_fn*)

Returns a new *Predicate* that combines this predicate with another predicate using a given function to combine the results.

```
>>> gt_2 = Predicate(lambda x: x > 2)
>>> even = Predicate(lambda x: x % 2 == 0)
>>> nand = lambda x, y: not (x and y)
>>> r = gt_2.combine(even, nand)
>>> map(r, range(1,5))
[True, True, True, False]
```

Specific Predicates

Several useful predefined predicates are provided by `qecc`.

class `qecc.`**SetMembershipPredicate**(*S*)

Given an iterable `S`, constructs a predicate that returns `True` if and only if its argument is in `S`.

```
>>> from qecc import SetMembershipPredicate
>>> p = SetMembershipPredicate(range(4))
>>> map(p, range(-1, 5))
[False, True, True, True, True, False]
```

class `qecc.`**PauliMembershipPredicate**(*S*, *ignore_phase=True*)

Given a set `S` of Pauli operators represented as *qecc.Pauli* instances, constructs a predicate that returns `True` for a Pauli `P` if and only if `P` is in `S`.

If the keyword argument `ignore_phase` is `True`, then the comparison to determine whether `P` is in `S` only considers the operator part of `P`.

In addition, utility functions are provided for constructing predicates based on commutation properties of the Pauli group.

`qecc.`**commutes_with**(**paulis*)

Returns a predicate that checks whether a Pauli `P` commutes with each of a given list of Pauli operators.

`qecc.`**in_group_generated_by**(**paulis*)

Returns a predicate that selects Pauli operators in the group generated by a given list of generators.

Usage Examples

Predicate functions can be used to quickly generate collections of `qecc.Pauli` operators having a given set of properties.

```
>>> from qecc import commutes_with, in_group_generated_by, pauli_group
>>> print filter(
...     commutes_with('XX', 'ZZ') & ~in_group_generated_by('XX'),
...     pauli_group(2)
...     )
[i^0 YY, i^0 ZZ]
```

Since searching in this way requires examining every element of a given iterator, it can be significantly faster to instead use constraint solvers such as those documented in *Constraint Solvers*.

1.8 Misc. Utility Functions and Classes

1.8.1 Matrix Manipulation

`qecc.`**`directsum`**`(`*A*, *B*`)`

Given two matrices A and B with two indices each, returns the direct sum $A \oplus B$.

Return type ndarray, shape (sA[0] + sB[0], sA[1] + sB[1])

Returns $A \oplus B$

`qecc.`**`parity`**`(`*bitarray*`)`

Parameters **`bitarray`** (`list`) – a list containing integers of value 0 or 1.

Returns True if bitarray is of odd parity, False if it is of even parity.

Return type bool

1.9 Exceptions and Warnings

1.9.1 `qecc.InvalidCliffordError` Reference

`class` `qecc.`**`InvalidCliffordError`**

We raise this exception wherever an automated procedure has produced a list of output Pauli matrices that does not commute as the result of an automorphism on the Paulis, or where some other idiocy has occurred.

1.10 Bibliography

CHAPTER 2

Introduction

QuaEC is a library for working with quantum error correction and fault-tolerance. In particular, QuaEC provides support for maniuplating Pauli and Clifford operators, as well as binary symplectic representations of each.

QuaEC is intended to provide easy, automated analysis of error-correcting protocols based on stabilizer codes. for example, one can define a stabilizer code from a pre-existing library, and produce an object representing a circuit to encode data into that code:

```
>>> import qecc as q
>>> perfect_code=q.StabilizerCode.perfect_5q_code()
>>> print perfect_code.encoding_cliffords().next().circuit_decomposition()
    CNOT    1 0
    CNOT    3 0
    CNOT    4 0
    CNOT    2 1
    CNOT    3 2
    CNOT    4 2
    CNOT    4 3
    H       0
    H       1
    H       2
    H       3
    H       4
    CZ      0 1
    CZ      0 2
    CZ      1 2
    CZ      0 3
    CZ      1 4
    CZ      3 4
    H       0
    H       1
    H       2
    H       3
    H       4
    H       4
```

(continues on next page)

(continued from previous page)

```
    SWAP     3 4
    H        4
    CNOT     0 4
    CNOT     0 3
    CNOT     0 2
    CNOT     0 1
    X        1
    X        2
    Z        3
    Y        4
```

CHAPTER 3

Getting Started with QuaEC

3.1 Obtaining QuaEC

Currently, QuaEC is hosted on GitHub. The latest unstable version of QuaEC is available for download as a ZIP there. Stable releases can be found on the downloads page, including installation packages for Windows and common Linux distributions.

QuaEC is available via PyPI as well. To obtain it, run `easy_install quaec` at the terminal or in the Windows command line.

3.2 Installation

Once you have obtained QuaEC, installation is straightforward using the included `setup.py` script or the installation packages.

To use `setup.py` on Unix-like systems, run the following commands from the command line:

```
$ cd /path/to/quaec/
$ sudo python setup.py install
```

To use `setup.py` on Windows, run `cmd.exe`, then enter the following commands:

```
C:\> cd C:\path\to\quaec\
C:\path\to\quaec\> python setup.py install
```

You may be prompted for permission by User Access Control, as the installer attempts to install QuaEC into the system-wide packages directory.

Once QuaEC has been installed, it is made available as the `qecc` package:

```
>>> import qecc as q
>>> print q.Pauli('XYZ', phase=2)
i^2 XYZ
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[AG04] Aaronson S. & Gottesman D. Improved simulation of stabilizer circuits. Phys. Rev. A 70, 052328 (2004). doi:10.1103/PhysRevA.70.052328.

[Got97] Gottesman, D. Stabilizer Codes and Quantum Error Correction. arXiv:quant-ph/9705052 (1997).

[CRSS96] Calderbank A.R., Rains E.M., Shor P.W. & Sloane N.J.A. Quantum Error Correction via Codes over GF(4). arXiv:quant-ph/9608006 (1996).

Index