---

description: | API documentation for modules: qfa_toolkit, qfa_toolkit.qiskit_converter, qfa_toolkit.qiskit_converter.qiskit_base, qfa_toolkit.qiskit_converter.qiskit_measure_many_quantum_finite_state_automaton, qfa_toolkit.qiskit_converter.qiskit_measure_once_quantum_finite_state_automaton, qfa_toolkit.qiskit_converter.utils, qfa_toolkit.quantum_finite_state_automaton, qfa_toolkit.quantum_finite_state_automaton.measure_many_quantum_finite_state_automaton, qfa_toolkit.quantum_finite_state_automaton.measure_once_quantum_finite_state_automaton, qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base, qfa_toolkit.quantum_finite_state_automaton.utils, qfa_toolkit.quantum_finite_state_automaton_language, qfa_toolkit.quantum_finite_state_automaton_language.measure_many_quantum_finite_state_automaton_language, qfa_toolkit.quantum_finite_state_automaton_language.measure_once_quantum_finite_state_automaton_language, qfa_toolkit.quantum_finite_state_automaton_language.quantum_finite_state_automaton_language_base, qfa_toolkit.recognition_strategy, qfa_toolkit.recognition_strategy.recognition_strategy.

lang: en

classoption: oneside geometry: margin=1in papersize: a4

linkcolor: blue links-as-notes: true ...

Namespace `qfa_toolkit` {#id}

Sub-modules

- qfa_toolkit.qiskit_converter
- qfa_toolkit.quantum_finite_state_automaton
- qfa_toolkit.quantum_finite_state_automaton_language
- qfa_toolkit.recognition_strategy

Module `qfa_toolkit.qiskit_converter` {#id}

Sub-modules

- qfa_toolkit.qiskit_converter.qiskit_base
- qfa_toolkit.qiskit_converter.qiskit_measure_many_quantum_finite_state_automaton
- qfa_toolkit.qiskit_converter.qiskit_measure_once_quantum_finite_state_automaton
- qfa_toolkit.qiskit_converter.utils

Module `qfa_toolkit.qiskit_converter.qiskit_base` {#id}

Classes

Class `QiskitQuantumFiniteStateAutomaton` {#id}

```
class QiskitQuantumFiniteStateAutomaton(

qfa: qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_b
```

```
)
```

Helper class that provides a standard way to create an ABC using inheritance.

-- Properties -- qfa: QuantumFiniteStateAutomaton size: int mapping: dict[int, int] defined_states: set[int] undefined_states: set[int] circuit: list[QuantumCircuit]

Ancestors (in MRO)

- abc.ABC

Descendants

- qfa_toolkit.qiskit_converter.qiskit_measure_many_quantum_finite_state_automaton.QiskitMeasureManyQuantumFiniteStateAutomaton
- qfa_toolkit.qiskit_converter.qiskit_measure_once_quantum_finite_state_automaton.QiskitMeasureOnceQuantumFiniteStateAutomaton

Instance variables

Variable `alphabet` {#id}

Variable `defined_states` {#id}

Type: `set[int]`

Variable `reverse_mapping` {#id}

Type: `dict[int, int]`

Variable `states` {#id}

Variable `undefined_states` {#id}

Type: `set[int]`

Methods

Method `get_circuit_for_string` {#id}

```
def get_circuit_for_string(
    self,
    w: list[int]
)
```

Method `get_mapping` {#id}

```
def get_mapping(
    self
)
```

Method `get_size` {#id}

```
def get_size(
    self
)
```

Method `transitions_to_circuit` {#id}

```
def transitions_to_circuit(
    self,
    transitions
)
```

Module `qfa_toolkit.qiskit_converter.qiskit_measure_many_quantum_` {#id}

Classes

Class `QiskitMeasureManyQuantumFiniteStateAutomaton` {#id}

```
class QiskitMeasureManyQuantumFiniteStateAutomaton(


qfa: qfa_toolkit.quantum_finite_state_automaton.measure_many_quantum_finite_stat
    use_entropy_mapping: bool = True
)
```

Helper class that provides a standard way to create an ABC using inheritance.

-- Properties -- qfa: Mmqfa accepting_states: set[int] rejecting_states: set[int] circuit: list[QuantumCircuit] # indexed by the alphabet size: int # size of the qubit register for states mapping: dict[int, int] # mapping from qubit register index to qfa state

Ancestors (in MRO)

- qfa_toolkit.qiskit_converter.qiskit_base.QiskitQuantumFiniteStateAutomaton
- abc.ABC

Instance variables

Variable `halting_states` {#id}

Methods

Method `get_circuit_for_string` {#id}

```
def get_circuit_for_string(
    self,
    w: list[int]
```

```
)
```

Method `get_entropy_mapping` {#id}

```
def get_entropy_mapping(
    self
)
```

Method `get_mapping` {#id}

```
def get_mapping(
    self
)
```

Method `get_size` {#id}

```
def get_size(
    self
)
```

Module `qfa_toolkit.qiskit_converter.qiskit_measure_once_quantum_` {#id}

Classes

Class `QiskitMeasureOnceQuantumFiniteStateAutomaton` {#id}

```
class QiskitMeasureOnceQuantumFiniteStateAutomaton(


qfa: qfa_toolkit.quantum_finite_state_automaton.measure_once_quantum_finite_stat
    use_entropy_mapping: bool = True
)
```

Helper class that provides a standard way to create an ABC using inheritance.

-- Properties -- qfa: QuantumFiniteStateAutomaton size: int mapping: dict[int, int] defined_states: set[int] undefined _states: set[int] circuit: list[QuantumCircuit]

Ancestors (in MRO)

- qfa_toolkit.qiskit_converter.qiskit_base.QiskitQuantumFiniteStateAutomaton
- abc.ABC

Methods

Method `get_circuit_for_string` {#id}

```
def get_circuit_for_string(
    self,
    w: list[int]
)
```

Method `get_entropy_mapping` {#id}

```
def get_entropy_mapping(
    self
)
```

Method `get_mapping` {#id}

```
def get_mapping(
    self
)
```

Module `qfa_toolkit.qiskit_converter.utils` {#id}

Functions

Function `unitary_matrix_to_circuit` {#id}

```
def unitary_matrix_to_circuit(
    unitary_matrix,
    label=None
)
```

Use qiskit unitary gate to convert unitary matrix to circuit

Module `qfa_toolkit.quantum_finite_state_automaton` {#id}

Sub-modules

- qfa_toolkit.quantum_finite_state_automaton.measure_many_quantum_finite_state_automaton
- qfa_toolkit.quantum_finite_state_automaton.measure_once_quantum_finite_state_automaton
- qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base
- qfa_toolkit.quantum_finite_state_automaton.utils

Module `qfa_toolkit.quantum_finite_state_automaton.measure_many_q`

{#id}

Classes

Class `MeasureManyQuantumFiniteStateAutomaton` {#id}

```
class MeasureManyQuantumFiniteStateAutomaton(

transition: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]],

accepting_states: numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]],

rejecting_states: numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.QuantumFiniteStateAutomatonBase
- abc.ABC

Class variables

Variable `start_of_string` {#id}

Type: `int`

Instance variables

Variable `halting_states` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Variable `non_halting_states` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Variable `observable` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Static methods

`Method linear_combination` {#id}

```
def linear_combination(
    *mmqfas: ~MmqfaT,
    coefficients: Optional[list[float]] = None
```

```
) -> ~MmqfaT
```

Returns the linear combination of the measure-once quantum finite automata.

For quantum finite automata M, N and 0 <= c <= 1, the linear combination M' is an mmqfa such that M'(w) = c * M(w) + (1 - c) * N(w) for all w.

Alberto Bertoni, Carlo Mereghetti, and Beatrice Palano. 2003. Quantum Computing: 1-Way Quantum Automata. In Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04).

Methods

Method `complement` {#id}

```
def complement(
    self: ~MmqfaT
) -> ~MmqfaT
```

Returns the complement of the quantum finite automaton.

For a quantum finite automaton M, the complement is defined as the quantum finite automaton M' such that M'(w) = 1 - M(w) for all w.

Alberto Bertoni, Carlo Mereghetti, and Beatrice Palano. 2003. Quantum Computing: 1-Way Quantum Automata. In Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04).

Method `counter_example` {#id}

```
def counter_example(
    self,
    other: ~MmqfaT
) -> Optional[list[int]]
```

Returns a counter example of the equivalence of the measure-many quantum finite automaton.

For quantum finite automata M and M', the counter example is defined as a word w such that M(w) != M'(w).

Method `equivalence` {#id}

```
def equivalence(
    self,
    other: ~MmqfaT
) -> bool
```

Returns whether the measure-many quantum finite automaton is equal.

For quantum finite automata M and M', the equivalence is defined as whether M(w) = M'(w) for all w.

See also counter_example().

Method `intersection` {#id}

```
def intersection(
    self: ~MmqfaT,
    other: ~MmqfaT
```

```
) -> ~MmqfaT
```

Returns the intersection of two measure-many quantum finite automata.

For a quantum finite automaton M and N, the intersection is defined as the quantum finite automaton M' such that M '(w) = M(w) * N(w) for all w.

Generally, MMQFA is not closed under the intersection. However, end-decisive MMQFAs with pure states are closed under the intersection. Note that this is not a necessary condition.

Maria Paola Bianchi and Beatrice Palano. 2010. Behaviours of Unary Quantum Automata. Fundamenta Informaticae.

Raises: NotClosedUnderOperationError

Method `inverse_homomorphism` {#id}

```
def inverse_homomorphism(
    self: ~MmqfaT,
    phi: list[list[int]]
) -> ~MmqfaT
```

Returns the inverse homomorphism of the measure-many quantum finite automaton.

For a quantum finite automaton M and a homomorphism phi, the inverse homomorphism M' of M with respect to phi is an MMQFA M' such that M'(w) = M(phi(w)).

Alex Brodsky and Nicholas Pippenger 2002. Characterizations of 1-way Quantum Finite Automata. SIAM Journal on Computing.

Method `is_co_end_decisive` {#id}

```
def is_co_end_decisive(
    self
) -> bool
```

Returns whether the quantum finite automaton is co-end-decisive.

A quantum finite automaton is end-decisive if it rejects only after read end-of-string.

Alex Brodsky and Nicholas Pippenger. 2002. Characterizations of 1-way Quantum Finite Automata. SIAM Journal on Computing.

Method `is_end_decisive` {#id}

```
def is_end_decisive(
    self
) -> bool
```

Returns whether the quantum finite automaton is end-decisive.

A quantum finite automaton is end-decisive if it accepts only after read end-of-string.

Alex Brodsky and Nicholas Pippenger. 2002. Characterizations of 1-way Quantum Finite Automata. SIAM Journal on Computing.

Method `step` {#id}

```
def step(
    self,


total_state: qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_aut
    c: int
) ·>

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `to_real_valued` {#id}

```
def to_real_valued(
    self: ~MmqfaT
) ·> ~MmqfaT
```

Method `union` {#id}

```
def union(
    self: ~MmqfaT,
    other: ~MmqfaT
) ·> ~MmqfaT
```

Returns the union of two measure-many quantum finite automata.

For a quantum finite automaton M and N, the union is defined as the quantum finite automaton M' such that 1 - M'(w) = (1 - M(w)) * (1 - N(w)) for all w.

Generally, MMQFA is not closed under the union. See intersection() for details.

Maria Paola Bianchi and Beatrice Palano. 2010. Behaviours of Unary Quantum Automata. Fundamenta Informaticae.

Raises: NotClosedUnderOperationError

Method `word_quotient` {#id}

```
def word_quotient(
    self: ~MmqfaT,
    w: list[int]
) ·> ~MmqfaT
```

Module `qfa_toolkit.quantum_finite_state_automaton.measure_once_` {#id}

Classes

Class `MeasureOnceQuantumFiniteStateAutomaton` {#id}

```
class MeasureOnceQuantumFiniteStateAutomaton(


transitions: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]],


accepting_states: numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.QuantumFiniteStateAutomatonBase
- abc.ABC

Class variables

Variable `start_of_string` {#id}

Type: `int`

Instance variables

Variable `observable` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Variable `rejecting_states` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Static methods

`Method linear_combination` {#id}

```
def linear_combination(
    *moqfas: ~MoqfaT,
    coefficients: Optional[list[float]] = None
) -> ~MoqfaT
```

Returns the linear combination of the measure-once quantum finite automata.

For quantum finite automata M, N and 0 <= c <= 1, the linear combination M' is an MOQFA such that M'(w) = c * M(w) + (1 - c) * N(w) for all w.

Alberto Bertoni, Carlo Mereghetti, and Beatrice Palano. 2003. Quantum Computing: 1-Way Quantum Automata. In Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04).

Methods

Method `bilinearize` {#id}

```
def bilinearize(
    self: ~MoqfaT
) -> ~MoqfaT
```

Method `bilinearized_call` {#id}

```
def bilinearized_call(
    self,
    w: list[int]
) -> float
```

Method `complement` {#id}

```
def complement(
    self: ~MoqfaT
) -> ~MoqfaT
```

Returns the complement of the measure-once quantum finite automaton.

For a quantum finite automaton M, the complement is defined as the quantum finite automaton M' such that M'(w) = 1 - M(w) for all w.

Alberto Bertoni, Carlo Mereghetti, and Beatrice Palano. 2003. Quantum Computing: 1-Way Quantum Automata. In Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04).

Method `counter_example` {#id}

```
def counter_example(
    self,
    other: ~MoqfaT
) -> Optional[list[int]]
```

Returns a counter example of the equivalence of the measure-once quantum finite automaton.

For quantum finite automata M and M', the counter example is defined as a word w such that M(w) != M'(w).

Lvzhou Li and Daowen Qiu. 2009. A note on quantum sequential machines. Theoretical Computer Science (TCS' 09).

Method `equivalence` {#id}

```
def equivalence(
    self,
    other: ~MoqfaT
) -> bool
```

Returns whether the two measure-once quantum finite automata are equal.

For quantum finite automata M and M', the equivalence is defined as whether M(w) = M'(w) for all w.

See also counter_example().

Method `intersection` {#id}

```
def intersection(
    self: ~MoqfaT,
    other: ~MoqfaT
) -> ~MoqfaT
```

Returns the mn-size intersection of the measure-once quantum finite automata.

For a quantum finite automaton M and N, the intersection, also known as Hadamard product, is defined as the quantum finite automaton M' such that M'(w) = M(w) * N(w) for all w.

Alberto Bertoni, Carlo Mereghetti, and Beatrice Palano. 2003. Quantum Computing: 1-Way Quantum Automata. In Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04).

Method `inverse_homomorphism` {#id}

```
def inverse_homomorphism(
    self: ~MoqfaT,
    phi: list[list[int]]
) -> ~MoqfaT
```

Returns the inverse homomorphism of the measure-once quantum finite automaton.

For a quantum finite automaton M and a homomorphism phi, the inverse homomorphism M' of M with respect to phi is an MOQFA M' such that M'(w) = M(phi(w)).

Cristopher Moore and James P. Crutchfield. 2000. Quantum Automata and Quantum Grammars. Theoretical Computer Science (TCS'00).

Method `step` {#id}

```
def step(
    self,

total_state: qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_aut
    c: int
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `to_bilinear` {#id}

```
def to_bilinear(
    self: ~MoqfaT
) -> ~MoqfaT
```

Returns the (n^2)-size bilinear form of the quantum finite automaton.

For a quantum finite automaton M, the bilinear form M' of M is an automaton such that M(w) is the sum of amplitude of the accepting states at the end of the computation of M'.

Cristopher Moore and James P. Crutchfield. 2000. Quantum Automata and Quantum Grammars. Theoretical

Computer Science (TCS'00).

Method `to_measure_many_quantum_finite_state_automaton` {#id}

```
def to_measure_many_quantum_finite_state_automaton(
    self
) ·>

qfa_toolkit.quantum_finite_state_automaton.measure_many_quantum_finite_state_aut
```

Method `to_real_valued` {#id}

```
def to_real_valued(
    self: ~MoqfaT
) ·> ~MoqfaT
```

Returns the 2n-size real-valued form of the quantum finite automaton.

Cristopher Moore and James P. Crutchfield. 2000. Quantum Automata and Quantum Grammars. Theoretical Computer Science (TCS'00).

Method `to_stochastic` {#id}

```
def to_stochastic(
    self: ~MoqfaT
) ·> ~MoqfaT
```

Returns the 2(n^2)-size stochastic form of the quantum finite automaton.

For a quantum finite automaton M, the bilinear form M' of M is an automaton such that M(w) is the sum of amplitude of the accepting states at the end of the computation of M'. Furthermore, the transitions of the M' is real-valued.

Cristopher Moore and James P. Crutchfield. 2000. Quantum Automata and Quantum Grammars. Theoretical Computer Science (TCS'00).

Method `to_without_final_transition` {#id}

```
def to_without_final_transition(
    self: ~MoqfaT
) ·> ~MoqfaT
```

Returns the quantum finite automaton without the final transition.

Alex Brodsky, and Nicholas Pippenger. 2002. Characterizations of 1-Way Quantum Finite Automata. SIAM Jornal on Computing 31.5.

Method `to_without_initial_transition` {#id}

```
def to_without_initial_transition(
    self: ~MoqfaT
) ·> ~MoqfaT
```

Returns the quantum finite automaton without the initial transition.

Alex Brodsky, and Nicholas Pippenger. 2002. Characterizations of 1-Way Quantum Finite Automata. SIAM Jornal on Computing 31.5.

Method `union` {#id}

```
def union(
    self: ~MoqfaT,
    other: ~MoqfaT
) ‑> ~MoqfaT
```

Returns the mn-size union of the two m- and n-size measure-once quantum finite automata.

For a quantum finite automaton M and N, the union is defined as the quantum finite automaton M' such that 1 - M'(w) = (1 - M(w)) * (1 - N(w)) for all w.

See also intersection().

Method `word_quotient` {#id}

```
def word_quotient(
    self: ~MoqfaT,
    w: list[int]
) ‑> ~MoqfaT
```

Returns the word quotient of the measure-once quantum finite automaton.

For a quantum finite automaton M and a word w, the word quotient M' of M with respect to u is an MOQFA M' such that M'(w) = M(uw) for all w.

Method `word_transition` {#id}

```
def word_transition(
    self,
    w: list[int]
) ‑>
numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
```

Module `qfa_toolkit.quantum_finite_state_automaton.quantum_finite` {#id}

Classes

Class `InvalidQuantumFiniteStateAutomatonError` {#id}

```
class InvalidQuantumFiniteStateAutomatonError(
    *args,
```

```
        **kwargs
    )
```

Common base class for all non-exit exceptions.

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Class `NotClosedUnderOperationException` {#id}

```
class NotClosedUnderOperationException(
    *args,
    **kwargs
)
```

Common base class for all non-exit exceptions.

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Class `QuantumFiniteStateAutomatonBase` {#id}

```
class QuantumFiniteStateAutomatonBase(


transitions: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- abc.ABC

Descendants

- qfa_toolkit.quantum_finite_state_automaton.measure_many_quantum_finite_state_automaton.MeasureManyQuantumFiniteStateAutomaton
- qfa_toolkit.quantum_finite_state_automaton.measure_once_quantum_finite_state_automaton.MeasureOnceQuantumFiniteStateAutomaton

Class variables

Variable `start_of_string` {#id}

Type: `int`

Instance variables

Variable `alphabet` {#id}

Type: `int`

Variable `end_of_string` {#id}

Type: `int`

Variable `final_transition` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]`

Variable `initial_transition` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]`

Variable `observable` {#id}

Type: `numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]`

Variable `states` {#id}

Type: `int`

Methods

Method `process` {#id}

```
def process(
    self,
    w: list[int],


total_state: Optional[qfa_toolkit.quantum_finite_state_automaton.quantum_finite_
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `step` {#id}

```
def step(
    self,


total_state: qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_aut
    c: int
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `string_to_tape` {#id}

```
def string_to_tape(
    self,
    string: list[int]
```

```
) -> list[int]
```

Class `TotalState` {#id}

```
class TotalState(


superposition_or_list: Union[numpy.ndarray[Any, numpy.dtype[numpy.complex128]],
    acceptance: float = 0,
    rejection: float = 0
)
```

Attila Kondacs and John Watros. On the power of quantum finite automata. 1997. 38th Annual Symposium on Foundations of Computer Science

Static methods

Method `initial` {#id}

```
def initial(
    states: int
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Methods

Method `apply` {#id}

```
def apply(
    self,


unitary: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `measure_by` {#id}

```
def measure_by(
    self,


observable: numpy.ndarray[typing.Any, numpy.dtype[numpy.bool_]]
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `normalized` {#id}

```
def normalized(
    self
) ->

qfa_toolkit.quantum_finite_state_automaton.quantum_finite_state_automaton_base.T
```

Method `to_tuple` {#id}

```
def to_tuple(
    self
) ->

tuple[numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]], float, float]
```

Module `qfa_toolkit.quantum_finite_state_automaton.utils` {#id}

Functions

Function `direct_sum` {#id}

```
def direct_sum(


u: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]],


v: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
) ->
numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
```

Returns the direct sum of two matrices.

Direct sum of U, V: (U, V) |-> [U 0; 0 V]

Function `get_real_valued_transition` {#id}

```
def get_real_valued_transition(


transition: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]
```

Function get_transition_from_initial_to_superposition {#id}

```
def get_transition_from_initial_to_superposition(


superposition: numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
) ·>
numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
```

Function mapping_to_transition {#id}

```
def mapping_to_transition(
    mapping: dict[int, int]
) ·>
numpy.ndarray[typing.Any, numpy.dtype[numpy.complex128]]
```

Module qfa_toolkit.quantum_finite_state_automaton_language {#id}

Sub-modules

- qfa_toolkit.quantum_finite_state_automaton_language.measure_many_quantum_finite_state_automaton_language
- qfa_toolkit.quantum_finite_state_automaton_language.measure_once_quantum_finite_state_automaton_language
- qfa_toolkit.quantum_finite_state_automaton_language.quantum_finite_state_automaton_language_base

Module qfa_toolkit.quantum_finite_state_automaton_language.measu {#id}

Classes

Class MeasureManyQuantumFiniteStateAutomatonLanguage {#id}

```
class MeasureManyQuantumFiniteStateAutomatonLanguage(
    quantum_finite_state_automaton: ~QfaT,
    strategy: ~RecognitionStrategyT
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- qfa_toolkit.quantum_finite_state_automaton_language.quantum_finite_state_automaton_language_base.QuantumFiniteStateAutomatonLanguageBase
- abc.ABC
- typing.Generic

Static methods

`Method from_unary_finite` {#id}

```
def from_unary_finite(
    ks: list[int],
    params: Optional[tuple[float, float]] = None
) ->

MeasureManyQuantumFiniteStateAutomatonLanguage[NegOneSided]
```

`Method from_unary_singleton` {#id}

```
def from_unary_singleton(
    k: int,
    params: Optional[tuple[float, float]] = None
) ->

MeasureManyQuantumFiniteStateAutomatonLanguage[NegOneSided]
```

Methods

Method `intersection` {#id}

```
def intersection(
    self: ~MmqflT,
    other: ~MmqflT
) -> ~MmqflT
```

Method `union` {#id}

```
def union(
    self: ~MmqflT,
    other: ~MmqflT
) -> ~MmqflT
```

Module `qfa_toolkit.quantum_finite_state_automaton_language.meas` {#id}

Classes

Class `MeasureOnceQuantumFiniteStateAutomatonLanguage` {#id}

```
class MeasureOnceQuantumFiniteStateAutomatonLanguage(
    quantum_finite_state_automaton: ~QfaT,
    strategy: ~RecognitionStrategyT
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- qfa_toolkit.quantum_finite_state_automaton_language.quantum_finite_state_automaton_language_base.QuantumFiniteStateAutomatonLanguageBase
- abc.ABC
- typing.Generic

Static methods

Method `from_modulo` {#id}

```
def from_modulo(
    n: int
) -> ~MoqflT
```

Create a quantum finite state automaton that recognizes the language of strings whose length is divisible by n.

Method `from_modulo_prime` {#id}

```
def from_modulo_prime(
    p: int,
    seed: int = 42
) -> ~MoqflT
```

Create a quantum finite state automaton that recognizes the language of strings whose length is divisible by p.

TODO: Add references.

Methods

Method `intersection` {#id}

```
def intersection(
    self,
    other: ~MoqflT
) -> ~MoqflT
```

Method `inverse_homomorphism` {#id}

```
def inverse_homomorphism(
```

```
    self,
    phi: list[list[int]]
) ·> ~MoqflT
```

Method `union` {#id}

```
def union(
    self,
    other: ~MoqflT
) ·> ~MoqflT
```

Method `word_quotient` {#id}

```
def word_quotient(
    self,
    w: list[int]
) ·> ~MoqflT
```

Module `qfa_toolkit.quantum_finite_state_automaton_language.quant` {#id}

Classes

Class `QuantumFiniteStateAutomatonLanguageBase` {#id}

```
class QuantumFiniteStateAutomatonLanguageBase(
    quantum_finite_state_automaton: ~QfaT,
    strategy: ~RecognitionStrategyT
)
```

Helper class that provides a standard way to create an ABC using inheritance.

Ancestors (in MRO)

- abc.ABC
- typing.Generic

Descendants

- qfa_toolkit.quantum_finite_state_automaton_language.measure_many_quantum_finite_state_automaton_language.MeasureManyQuantumFiniteStateAutomatonLanguage
- qfa_toolkit.quantum_finite_state_automaton_language.measure_once_quantum_finite_state_automaton_language.MeasureOnceQuantumFiniteStateAutomatonLanguage

Instance variables

Variable `alphabet` {#id}

Type: `int`

Variable `end_of_string` {#id}

Type: `int`

Variable `start_of_string` {#id}

Type: `int`

Methods

Method `enumerate` {#id}

```
def enumerate(
    self
) -> Iterator[list[int]]
```

Method `enumerate_length_less_than_n` {#id}

```
def enumerate_length_less_than_n(
    self,
    n: int
) -> Iterator[list[int]]
```

Method `enumerate_length_n` {#id}

```
def enumerate_length_n(
    self,
    n: int
) -> Iterator[list[int]]
```

Module `qfa_toolkit.recognition_strategy` {#id}

Sub-modules

- qfa_toolkit.recognition_strategy.recognition_strategy

Module `qfa_toolkit.recognition_strategy.recognition_strategy` {#id}

Classes

Class `CutPoint` {#id}

```
class CutPoint(
    probability: float
)
```

Ancestors (in MRO)

- qfa_toolkit.recognition_strategy.recognition_strategy.RecognitionStrategy

Class `IsolatedCutPoint` {#id}

```
class IsolatedCutPoint(
    threshold: float,
    epsilon: float
)
```

Michael O. Rabin, Probabilistic automata, Information and Control, Volume 6, Issue 3, 1963, Pages 230-245, ISSN 0019-9958, https://doi.org/10.1016/S0019-9958(63)90290-0.

Ancestors (in MRO)

- qfa_toolkit.recognition_strategy.recognition_strategy.RecognitionStrategy

Instance variables

Variable `epsilon` {#id}

Type: `float`

Variable `threshold` {#id}

Type: `float`

Class `NegativeOneSidedBoundedError` {#id}

```
class NegativeOneSidedBoundedError(
    epsilon: float
)
```

Ancestors (in MRO)

- qfa_toolkit.recognition_strategy.recognition_strategy.RecognitionStrategy

Instance variables

Variable `epsilon` {#id}

Type: `float`

Class `PositiveOneSidedBoundedError` {#id}

```
class PositiveOneSidedBoundedError(
    epsilon: float
)
```

Ancestors (in MRO)

- qfa_toolkit.recognition_strategy.recognition_strategy.RecognitionStrategy

Instance variables

Variable `epsilon` {#id}

Type: `float`

Class `RecognitionStrategy` {#id}

```
class RecognitionStrategy(
    reject_upperbound: float,
    accept_lowerbound: float,
    reject_inclusive: bool = False,
    accept_inclusive: bool = False
)
```

Descendants

- qfa_toolkit.recognition_strategy.recognition_strategy.CutPoint
- qfa_toolkit.recognition_strategy.recognition_strategy.IsolatedCutPoint
- qfa_toolkit.recognition_strategy.recognition_strategy.NegativeOneSidedBoundedError
- qfa_toolkit.recognition_strategy.recognition_strategy.PositiveOneSidedBoundedError

Class variables

Variable `Result` {#id}

An enumeration.
