qnet Documentation

Release 1.4.3

Nikolas Tezak and Michael Goerz

Mar 09, 2017

Contents

1 Installation/Setup **3**
- 1.1 Dependencies . . . 3
- 1.2 Installation/Configuration . . . 4
- 1.3 gEDA . . . 4

2 Symbolic Algebra **5**
- 2.1 The Abstract Algebra module . . . 5
- 2.2 Hilbert Space Algebra . . . 6
- 2.3 The Operator Algebra module . . . 7
- 2.4 The Circuit Algebra module . . . 10
- 2.5 The Super-Operator Algebra module . . . 13
- 2.6 The State (Ket-) Algebra module . . . 14

3 Properties and Simplification of Circuit Algebraic Expressions **15**
- 3.1 Permutation objects . . . 17
- 3.2 Permutations and Concatenations . . . 18
- 3.3 Feedback of a concatenation . . . 20
- 3.4 Feedback of a series . . . 21

4 Circuit Component Definition **25**
- 4.1 A simple example . . . 26
- 4.2 Creating custom component symbols for `gschem` . . . 27

5 Schematic Capture **29**

6 Netlisting **31**
- 6.1 Using `gnetlist` . . . 31
- 6.2 The QHDL Syntax . . . 31

7 Parsing QHDL **41**

8 Symbolic Analysis and Simulation **43**
- 8.1 Symbolic Analysis of the Pseudo NAND gate and the Pseudo NAND SR-Latch . . . 43
- 8.2 Numerical Analysis via QuTiP . . . 47

9 References **53**

10 API **55**

10.1 qnet package 55

11 Indices and tables **173**

Bibliography **175**

Python Module Index **177**

The QNET package is a set of tools to aid in the design and analysis of photonic circuit models, but it features a flexible symbolic algebra module that can be applied in a more general setting. Our proposed Quantum Hardware Description Language *[QHDL]* serves to describe a circuit topology and specification of a larger entity in terms of parametrizable subcomponents. By design this is analogous to the specification of electric circuitry using the structural description elements of VHDL or Verilog.

The physical systems that can be modeled within the framework include quantum optical experiments that can be described as nodes with internal degrees of freedom such as interacting quantum harmonic oscillators and/or N-level quantum systems that, in turn are coupled to a finite number of bosonic quantum fields. Furthermore, the formalism applies also to superconducting microwave circuit (Circuit QED) systems.

For a rigorous introduction to the underlying mathematical physics we refer to the original treatment of Gough and James *[GoughJames08]*, *[GoughJames09]* and the references given therein.

The main components of this package are:

1. A symbolic computer algebra package `qnet.algebra` for Hilbert Space quantum mechanical operators, the Gough-James circuit algebra and also an algebra for Hilbert space states and Super-operators.
2. The QHDL language definition and parser `qnet.qhdl` including a front-end located at `bin/parse_qhdl.py` that can convert a QHDL-file into a circuit component library file.
3. A library of existing primitive or composite circuit components `qnet.circuit_components` that can be embedded into a new circuit definition.

In practice one might want to use these to:

1. Define and specify your basic circuit component model and create a library file, *Circuit Component Definition*
2. Use `gschem` (of gEDA) to graphically design a circuit model, *Schematic Capture*
3. Export the schematic to QHDL using `gnetlist` (also part of gEDA) or directly write a QHDL file, *Netlisting*
4. Parse the QHDL-circuit definition file into a Python circuit library component using the parser front-end `bin/parse_qhdl.py`, *Parsing QHDL*
5. Analyze the model analytically using our symbolic algebra and/or numerically using QuTiP, *Symbolic Algebra*, *Symbolic Analysis and Simulation*

This package is still work in progress and as it is currently being developed by a single developer (interested in helping?), documentation and comprehensive testing code are still somewhat lacking. Any contributions, bug reports and general feedback from end-users would be highly appreciated. If you have found a bug, it would be extremely helpful if you could try to write a minimal code example that reproduces the bug. Feature requests will definitely be considered. Higher priority will be given to things that many people ask for and that can be implemented efficiently.

To learn of how to carry out each of these steps, we recommend looking at the provided examples and reading the relevant sections in the QNET manual. Also, if you want to implement and add your own primitive device models, please consult the QNET manual.

Contents:

CHAPTER 1

Installation/Setup

Dependencies

In addition to these core components, the software uses the following existing software packages:

0. Python version 3.3 or higher. Python 2 is no longer supported.
1. The gEDA toolsuite for its visual tool `gschem` for the creation of circuits end exporting these to QHDL `gnetlist`. We have created device symbols for our primitive circuit components to be used with `gschem` and we have included our own `gnetlist` plugin for exporting to QHDL.
2. The SymPy symbolic algebra Python package to implement symbolic 'scalar' algebra, i.e. the coefficients of state, operator or super-operator expressions can be symbolic SymPy expressions as well as pure python numbers.
3. The QuTiP python package as an extremely useful, efficient and full featured numerical backend. Operator expressions where all symbolic scalar parameters have been replaced by numeric ones, can be converted to (sparse) numeric matrix representations, which are then used to solve for the system dynamics using the tools provided by QuTiP.
4. The PyX python package for visualizing circuit expressions as box/flow diagrams.
5. The SciPy and NumPy packages (needed for QuTiP but also by the `qnet.algebra` package)
6. The PLY python package as a dependency of our Python Lex/Yacc based QHDL parser.

A convenient way of obtaining Python as well as some of the packages listed here (SymPy, SciPy, NumPy, PLY) is to download the Enthought Python Distribution (EPD) or Anaconda which are both free for academic use. A highly recommended way of working with QNET and QuTiP and just scientific python codes in action is to use the excellent IPython shell which comes both with a command-line interface as well as a very polished browser-based notebook interface.

Installation/Configuration

To install QNET you need a working Python installation as well as pip which comes pre-installed with both the Enthought Python distribution and Anaconda. If you have already installed PyX just run: Run:

```
pip install QNET
```

If you still need to install PyX, run:

```
pip install --process-dependency-links QNET
```

gEDA

Setting up gEDA/gschem/gnetlist is a bit more involved. If you are using Linux or OSX, geda is available via common package managers such as *port* and *homebrew* on OSX or apt for Linux.

To configure interoperability with QNET/QHDL this you will have to locate the installation directory of QNET. This can easily be found by running:

```
python -c "import qnet, os; print(os.path.join(*os.path.dirname(qnet.__file__).split(
↪'/')[:-1]))"
```

In BASH you can just run:

```
QNET=$(python -c "import qnet, os; print(os.path.join(*os.path.dirname(qnet.__file__).
↪split('/')[:-1]))")
```

to store this path in a shell variable named QNET. To configure gEDA to include our special quantum circuit component symbols you will need to copy the following configuration files from the `$QNET/gEDA_support/config` directory to the `$HOME/.gEDA` directory:

- `~/.gEDA/gafrc`
- `~/.gEDA/gschemrc`

Then install the QHDL netlister plugin within gEDA by creating a symbolic link (or copy the file there)

```
ln -s $QNET/gEDA_support/gnet-qhdl.scm  /path/to/gEDA_resources_folder/scheme/gnet-
↪qhdl.scm
```

Note that you should replace "/path/to/gEDA_resources_folder" with the full path to the gEDA resources directory!

in my case that path is given by `/opt/local/share/gEDA`, but in general simply look for the gEDA-directory that contains the file named `system-gafrc`.

CHAPTER 2

Symbolic Algebra

The Abstract Algebra module

> **Warning:** This overview is currently not up to date with respect to the latest development version of QNET. Please refer to the *API* instead.

The module features generic classes for encapsulating expressions and operations on expressions. It also includes some basic pattern matching and expression rewriting capabilities.

The most important classes to derive from for implementing a custom 'algebra' are *qnet.algebra.abstract_algebra.Expression* and *qnet.algebra.abstract_algebra.Operation*, where the second is actually a subclass of the first.

The `Operation` class should be subclassed to implement any structured expression type that can be specified in terms of a *head* and a (finite) sequence of *operands*:

```
Head(op1, op1, ..., opN)
```

An operation is assumed to have immutable operands, i.e., if one wishes to change the operands of an `Operation`, one rather creates a new Operation with modified Operands.

Defining `Operation` subclasses

The single most important method of the `Operation` class is the `qnet.algebra.abstract_algebra.Operation.create()` classmethod.

Automatic expression rewriting by modifying/decorating the `qnet.algebra.abstract_algebra.Operation.create()` method

A list of class decorators:

- *qnet.algebra.abstract_algebra.assoc()*

- *qnet.algebra.abstract_algebra.idem()*
- *qnet.algebra.abstract_algebra.orderby()*
- *qnet.algebra.abstract_algebra.filter_neutral()*
- qnet.algebra.abstract_algebra.check_signature()
- *qnet.algebra.abstract_algebra.match_replace()*
- *qnet.algebra.abstract_algebra.match_replace_binary()*

Pattern matching

The qnet.algebra.abstract_algebra.Wildcard class.

The qnet.algebra.abstract_algebra.match() function.

For a relatively simple example of how an algebra can be defined, see the Hilbert space algebra defined in *qnet.algebra.hilbert_space_algebra*.

Hilbert Space Algebra

This covers only finite dimensional or countably infinite dimensional Hilbert spaces.

The basic abstract class that features all properties of Hilbert space objects is given by: *qnet.algebra.hilbert_space_algebra.HilbertSpace*. Its most important subclasses are:

- local/primitive degrees of freedom (e.g. a single multi-level atom or a cavity mode) are described by a *qnet.algebra.hilbert_space_algebra.LocalSpace*. Every local space is identified by
- composite tensor product spaces are given by instances of the *qnet.algebra.hilbert_space_algebra.ProductSpace* class.
- the *qnet.algebra.hilbert_space_algebra.TrivialSpace* represents a *trivial*[1] Hilbert space $\mathcal{H}_0 \simeq \mathbb{C}$
- the *qnet.algebra.hilbert_space_algebra.FullSpace* represents a Hilbert space that includes all possible degrees of freedom.

Examples

A single local space can be instantiated in several ways. It is most convenient to use the qnet.algebra.hilbert_space_algebra.local_space() method:

```
>>> local_space(1)
    LocalSpace(1, '')
```

This method also allows for the specification of the dimension of the local degree of freedom's state space:

```
>>> s = local_space(1, dimension = 10)
>>> s
    LocalSpace(1, '')
>>> s.dimension
    10
```

[1] *trivial* in the sense that $\mathcal{H}_0 \simeq \mathbb{C}$, i.e., all states are multiples of each other and thus equivalent.

```
>>> s.basis
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Alternatively, one can pass a sequence of `basis` state labels instead of the `dimension` argument:

```
>>> lambda_atom_space = local_space('las', basis = ('e', 'h', 'g'))
>>> lambda_atom_space
    LocalSpace('las', '')
>>> lambda_atom_space.dimension
    3
>>> lambda_atom_space.basis
    ('e', 'h', 'g')
```

Finally, one can pass a `namespace` argument, which is useful if one is working with multiple copies of identical systems, e.g. if one instantiates multiple copies of a particular circuit component with internal degrees of freedom:

```
>>> s_q1 = local_space('s', namespace = 'q1', basis = ('g', 'h'))
>>> s_q2 = local_space('s', namespace = 'q2', basis = ('g', 'h'))
>>> s_q1
    LocalSpace('s', 'q1')
>>> s_q2
    LocalSpace('s', 'q2')
>>> s_q1 * s_q2
    ProductSpace(LocalSpace('s', 'q1'), LocalSpace('s', 'q2'))
```

The default `namespace` is the empty string `''`. Here, we have already seen the simplest way to create a tensor product of spaces:

```
>>> local_space(1) * local_space(2)
    ProductSpace(LocalSpace(1, ''), LocalSpace(2, ''))
```

Note that this tensor product is *commutative*

```
>>> local_space(2) * local_space(1)
    ProductSpace(LocalSpace(1, ''), LocalSpace(2, ''))
>>> local_space(2) * local_space(1) == local_space(1) * local_space(2)
    True
```

and *associative*

```
>>> (local_space(1) * local_space(2)) * local_space(3)
    ProductSpace(LocalSpace('1', ''), LocalSpace('2', ''), LocalSpace('3', ''))
```

The Operator Algebra module

This module features classes and functions to define and manipulate symbolic Operator expressions. Operator expressions are constructed from sums (*qnet.algebra.operator_algebra.OperatorPlus*) and products (*qnet.algebra.operator_algebra.OperatorTimes*) of some basic elements, most importantly *local* operators, such as the annihilation (*qnet.algebra.operator_algebra.Destroy*) and creation (*qnet.algebra.operator_algebra.Create*) operators $a_s, a_s^\dagger$ of a quantum harmonic oscillator degree of freedom s. Further important elementary local operators are the switching operators $\sigma_{jk}^s := |j\rangle_s \langle k|_s$ (*qnet.algebra.operator_algebra.LocalSigma*). Each operator has an associated *qnet.algebra.operator_algebra.Operator.space* property which gives the Hilbert space (cf *qnet.algebra.hilbert_space_algebra.HilbertSpace*) on which it acts *non-trivially*. We don't explicitly distinguish

between *tensor*-products $X_s \otimes Y_r$ of operators on different degrees of freedom s, r (which we designate as *local* spaces) and *operator-composition*-products $X_s \cdot Y_s$ of operators acting on the same degree of freedom s. Conceptionally, we assume that each operator is always implicitly tensored with identity operators acting on all un-specified degrees of freedom. This is typically done in the physics literature and only plays a role when tansforming to a numerical representation of the problem for the purpose of simulation, diagonalization, etc.

All Operator classes

A complete list of all local operators is given below:

- Harmonic oscillator mode operators $a_s, a_s^\dagger$ (cf `qnet.algebra.operator_algebra.Destroy`, `qnet.algebra.operator_algebra.Create`)
- σ-switching operators $\sigma_{jk}^s := |j\rangle_s \langle k|_s$ (cf `qnet.algebra.operator_algebra.LocalSigma`)
- coherent displacement operators $D_s(\alpha) := \exp\left(\alpha a_s^\dagger - \alpha^* a_s\right)$ (cf `qnet.algebra.operator_algebra.Displace`)
- phase operators $P_s(\phi) := \exp\left(i\phi a_s^\dagger a_s\right)$ (cf `qnet.algebra.operator_algebra.Phase`)
- squeezing operators $S_s(\eta) := \exp\left[\frac{1}{2}\left(\eta {a_s^\dagger}^2 - \eta^* a_s^2\right)\right]$ (cf `qnet.algebra.operator_algebra.Squeeze`)

Furthermore, there exist symbolic representations for constants and symbols:

- the identity operator (cf `qnet.algebra.operator_algebra.IdentityOperator`)
- and the zero operator (cf `qnet.algebra.operator_algebra.ZeroOperator`)
- an arbitrary operator symbol (cf `qnet.algebra.operator_algebra.OperatorSymbol`)

Finally, we have the following Operator operations:

- sums of operators $X_1 + X_2 + \dots + X_n$ (cf `qnet.algebra.operator_algebra.OperatorPlus`)
- products of operators $X_1 X_2 \cdots X_n$ (cf `qnet.algebra.operator_algebra.OperatorTimes`)
- the Hilbert space adjoint operator $X^\dagger$ (cf `qnet.algebra.operator_algebra.Adjoint`)
- scalar multiplication λX (cf `qnet.algebra.operator_algebra.ScalarTimesOperator`)
- **pseudo-inverse of operators** X^+ **satisfying** $XX^+X = X$ **and** $X^+XX^+ = X^+$ **as well** as $(X^+X)^\dagger = X^+X$ and $(XX^+)^\dagger = XX^+$ (cf `qnet.algebra.operator_algebra.PseudoInverse`)
- **the kernel projection operator** $\mathcal{P}_{\mathrm{Ker}X}$ **satisfying both** $X\mathcal{P}_{\mathrm{Ker}X} = 0$ and $X^+X = 1 - \mathcal{P}_{\mathrm{Ker}X}$ (cf `qnet.algebra.operator_algebra.NullSpaceProjector`)
- Partial traces over Operators $\mathrm{Tr}_s X$ (cf `qnet.algebra.operator_algebra.OperatorTrace`)

For a list of all properties and methods of an operator object, see the documentation for the basic `qnet.algebra.operator_algebra.Operator` class.

Examples

Say we want to write a function that constructs a typical Jaynes-Cummings Hamiltonian

$$H = \Delta\sigma^\dagger\sigma + \Theta a^\dagger a + ig(\sigma a^\dagger - \sigma^\dagger a) + i\epsilon(a - a^\dagger)$$

for a given set of numerical parameters:

```
def H_JaynesCummings(Delta, Theta, epsilon, g, namespace = ''):

    # create Fock- and Atom local spaces
    fock = local_space('fock', namespace = namespace)
    tls = local_space('tls', namespace = namespace, basis = ('e', 'g'))

    # create representations of a and sigma
    a = Destroy(fock)
    sigma = LocalSigma(tls, 'g', 'e')

    H = (Delta * sigma.dag() * sigma                        # detuning from atomic␣
↪resonance
        + Theta * a.dag() * a                              # detuning from cavity␣
↪resonance
        + I * g * (sigma * a.dag() - sigma.dag() * a)      # atom-mode coupling, I =␣
↪sqrt(-1)
        + I * epsilon * (a - a.dag()))                     # external driving␣
↪amplitude
    return H
```

Here we have allowed for a variable namespace which would come in handy if we wanted to construct an overall model that features multiple Jaynes-Cummings-type subsystems.

By using the support for symbolic `sympy` expressions as scalar pre-factors to operators, one can instantiate a Jaynes-Cummings Hamiltonian with symbolic parameters:

```
>>> Delta, Theta, epsilon, g = symbols('Delta, Theta, epsilon, g', real = True)
>>> H = H_JaynesCummings(Delta, Theta, epsilon, g)
>>> str(H)
    'Delta Pi_e^[tls] +  I*g ((a_fock)^* sigma_ge^[tls] - a_fock sigma_eg^[tls]) + ␣
↪I*epsilon ( - (a_fock)^* + a_fock) +  Theta (a_fock)^* a_fock'
```

```
>>> H.space
    ProductSpace(LocalSpace('fock', ''), LocalSpace('tls', ''))
```

or equivalently, represented in latex via `H.tex()` this yields:

$$\Delta \Pi_{\rm e}^{\rm tls} + \imath g \left(a_{\rm fock}^{\dagger} \sigma_{\rm g,e}^{\rm tls} - a_{\rm fock} \sigma_{\rm e,g}^{\rm tls}\right) + \imath \epsilon \left(-a_{\rm fock}^{\dagger} + a_{\rm fock}\right) + \Theta a_{\rm fock}^{\dagger} a_{\rm fock}$$

Operator products between commuting operators are automatically re-arranged such that they are ordered according to their Hilbert Space

```
>>> Create(2) * Create(1)
    OperatorTimes(Create(1), Create(2))
```

There are quite a few built-in replacement rules, e.g., mode operators products are normally ordered:

```
>>> Destroy(1) * Create(1)
    1 + Create(1) * Destroy(1)
```

Or for higher powers one can use the `expand()` method:

```
>>> (Destroy(1) * Destroy(1) * Destroy(1) * Create(1) * Create(1) * Create(1)).
↪expand()
    (6 + Create(1) * Create(1) * Create(1) * Destroy(1) * Destroy(1) * Destroy(1) + 9␣
↪* Create(1) * Create(1) * Destroy(1) * Destroy(1) + 18 * Create(1) * Destroy(1))
```

The Circuit Algebra module

In their works on networks of open quantum systems *[GoughJames08]*, *[GoughJames09]* Gough and James have introduced an algebraic method to derive the Quantum Markov model for a full network of cascaded quantum systems from the reduced Markov models of its constituents. A general system with an equal number n of input and output channels is described by the parameter triplet $(\mathbf{S}, \mathbf{L}, H)$, where H is the effective internal *Hamilton operator* for the system, $\mathbf{L} = (L_1, L_2, \dots, L_n)^T$ the *coupling vector* and $\mathbf{S} = (S_{jk})_{j,k=1}^n$ is the *scattering matrix* (whose elements are themselves operators). An element L_k of the coupling vector is given by a system operator that describes the system's coupling to the k-th input channel. Similarly, the elements S_{jk} of the scattering matrix are in general given by system operators describing the scattering between different field channels j and k. The only conditions on the parameters are that the hamilton operator is self-adjoint and the scattering matrix is unitary:

$$H^* = H \text{ and } \mathbf{S}^\dagger \mathbf{S} = \mathbf{S}\mathbf{S}^\dagger = \mathbf{1}_n.$$

We adhere to the conventions used by Gough and James, i.e. we write the imaginary unit is given by $i := \sqrt{-1}$, the adjoint of an operator A is given by A^*, the element-wise adjoint of an operator matrix $\mathbf{M}$ is given by $\mathbf{M}^\sharp$. Its transpose is given by $\mathbf{M}^T$ and the combination of these two operations, i.e. the adjoint operator matrix is given by $\mathbf{M}^\dagger = (\mathbf{M}^T)^\sharp = (\mathbf{M}^\sharp)^T$.

Fundamental Circuit Operations

The basic operations of the Gough-James circuit algebra are given by:

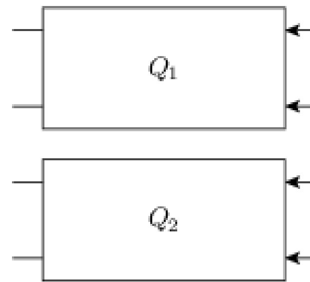


Fig. 2.1: $Q_1 \boxplus Q_2$

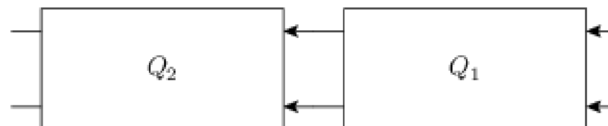


Fig. 2.2: $Q_2 \lhd Q_1$

In *[GoughJames09]*, Gough and James have introduced two operations that allow the construction of quantum optical 'feedforward' networks:

1. The *concatenation* product describes the situation where two arbitrary systems are formally attached to each other without optical scattering between the two systems' in- and output channels

$$(\mathbf{S}_1, \mathbf{L}_1, H_1) \boxplus (\mathbf{S}_2, \mathbf{L}_2, H_2) = \left(\begin{pmatrix} \mathbf{S}_1 & 0 \\ 0 & \mathbf{S}_2 \end{pmatrix}, \begin{pmatrix} \mathbf{L}_1 \\ \mathbf{L}_1 \end{pmatrix}, H_1 + H_2 \right)$$

Note however, that even without optical scattering, the two subsystems may interact directly via shared quantum degrees of freedom.

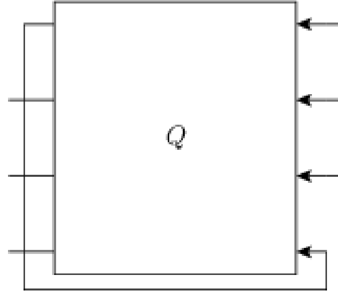


Fig. 2.3: $[Q]_{1\to 4}$

2. The *series* product is to be used for two systems $Q_j = (\mathbf{S}_j, \mathbf{L}_j, H_j)$, $j = 1, 2$ of equal channel number n where all output channels of Q_1 are fed into the corresponding input channels of Q_2

$$(\mathbf{S}_2, \mathbf{L}_2, H_2) \lhd (\mathbf{S}_1, \mathbf{L}_1, H_1) = \left(\mathbf{S}_2\mathbf{S}_1, \mathbf{L}_2 + \mathbf{S}_2\mathbf{L}_1, H_1 + H_2 + \Im\left\{\mathbf{L}_2^\dagger\mathbf{S}_2\mathbf{L}_1\right\}\right)$$

From their definition it can be seen that the results of applying both the series product and the concatenation product not only yield valid circuit component triplets that obey the constraints, but they are also associative operations.footnote{For the concatenation product this is immediately clear, for the series product in can be quickly verified by computing $(Q_1 \lhd Q_2) \lhd Q_3$ and $Q_1 \lhd (Q_2 \lhd Q_3)$. To make the network operations complete in the sense that it can also be applied for situations with optical feedback, an additional rule is required: The *feedback* operation describes the case where the k-th output channel of a system with $n \geq 2$ is fed back into the l-th input channel. The result is a component with $n-1$ channels:

$$\left[\;(\mathbf{S}, \mathbf{L}, H)\;\right]_{k\to l} = \left(\tilde{\mathbf{S}}, \tilde{\mathbf{L}}, \tilde{H}\right),$$

where the effective parameters are given by [GoughJames08]

$$\tilde{\mathbf{S}} = \mathbf{S}_{[k,l]} + \begin{pmatrix} S_{1l} \\ S_{2l} \\ \vdots \\ S_{k-1\,l} \\ S_{k+1\,l} \\ \vdots \\ S_{nl} \end{pmatrix} (1 - S_{kl})^{-1} \begin{pmatrix} S_{k1} & S_{k2} & \cdots & S_{kl-1} & S_{kl+1} & \cdots & S_{kn} \end{pmatrix},$$

$$\tilde{\mathbf{L}} = \mathbf{L}_{[k]} + \begin{pmatrix} S_{1l} \\ S_{2l} \\ \vdots \\ S_{k-1\,l} \\ S_{k+1\,l} \\ \vdots \\ S_{nl} \end{pmatrix} (1 - S_{kl})^{-1} L_k,$$

$$\tilde{H} = H + \Im\left\{\left[\sum_{j=1}^n L_j^* S_{jl}\right](1 - S_{kl})^{-1} L_k\right\}.$$

Here we have written $\mathbf{S}_{[k,l]}$ as a shorthand notation for the matrix $\mathbf{S}$ with the k-th row and l-th column removed and similarly $\mathbf{L}_{[k]}$ is the vector $\mathbf{L}$ with its k-th entry removed. Moreover, it can be shown that in the case of multiple feedback loops, the result is independent of the order in which the feedback operation is applied. Note however that some care has to be taken with the indices of the feedback channels when permuting the feedback operation.

The possibility of treating the quantum circuits algebraically offers some valuable insights: A given full-system triplet $(\mathbf{S}, \mathbf{L}, H)$ may very well allow for different ways of decomposing it algebraically into networks of physically realistic subsystems. The algebraic treatment thus establishes a notion of dynamic equivalence between potentially very different physical setups. Given a certain number of fundamental building blocks such as beamsplitters, phases and cavities, from which we construct complex networks, we can investigate what kinds of composite systems can be realized. If we also take into account the adiabatic limit theorems for QSDEs (cite Bouten2008a,Bouten2008) the set of physically realizable systems is further expanded. Hence, the algebraic methods not only facilitate the analysis of quantum circuits, but ultimately they may very well lead to an understanding of how to construct a general system $(\mathbf{S}, \mathbf{L}, H)$ from some set of elementary systems. There already exist some investigations along these lines for the particular subclass of *linear* systems (cite Nurdin2009a,Nurdin2009b) which can be thought of as a networked collection of quantum harmonic oscillators.

Representation as Python objects

This file features an implementation of the Gough-James circuit algebra rules as introduced in *[GoughJames08]* and *[GoughJames09]*. Python objects that are of the `qnet.algebra.circuit_algebra.Circuit` type have some of their operators overloaded to realize symbolic circuit algebra operations:

```
>>> A = CircuitSymbol('A', 2)
>>> B = CircuitSymbol('B', 2)
>>> A << B
    SeriesProduct(A, B)
>>> A + B
    Concatenation(A, B)
>>> FB(A, 0, 1)
    Feedback(A, 0, 1)
```

For a thorough treatment of the circuit expression simplification rules see *Properties and Simplification of Circuit Algebraic Expressions*.

Examples

Extending the JaynesCummings problem above to an open system by adding collapse operators $L_1 = \sqrt{\kappa}a$ and $L_2 = \sqrt{\gamma}\sigma$.

```
def SLH_JaynesCummings(Delta, Theta, epsilon, g, kappa, gamma, namespace = ''):

    # create Fock- and Atom local spaces
    fock = local_space('fock', namespace = namespace)
    tls = local_space('tls', namespace = namespace, basis = ('e', 'g'))

    # create representations of a and sigma
    a = Destroy(fock)
    sigma = LocalSigma(tls, 'g', 'e')

    # Trivial scattering matrix
    S = identity_matrix(2)

    # Collapse/Jump operators
    L1 = sqrt(kappa) * a                                # Decay of cavity mode␣
↪through mirror
    L2 = sqrt(gamma) * sigma                            # Atomic decay due to␣
↪spontaneous emission into outside modes.
    L = Matrix([[L1], \
                [L2]])
```

```
    # Hamilton operator
    H = (Delta * sigma.dag() * sigma                         # detuning from atomic␣
↪resonance
        + Theta * a.dag() * a                                # detuning from cavity␣
↪resonance
        + I * g * (sigma * a.dag() - sigma.dag() * a)        # atom-mode coupling, I =␣
↪sqrt(-1)
        + I * epsilon * (a - a.dag()))                       # external driving␣
↪amplitude

    return SLH(S, L, H)
```

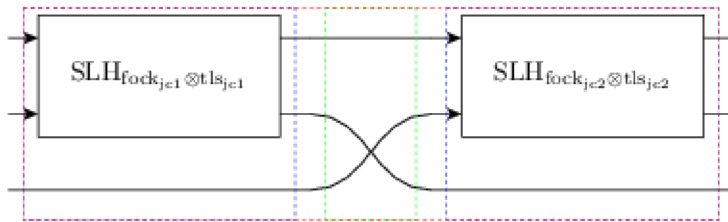
Consider now an example where we feed one Jaynes-Cummings system's output into a second one:

```
Delta, Theta, epsilon, g = symbols('Delta, Theta, epsilon, g', real = True)
kappa, gamma = symbols('kappa, gamma')

JC1 = SLH_JaynesCummings(Delta, Theta, epsilon, g, kappa, gamma, namespace = 'jc1')
JC2 = SLH_JaynesCummings(Delta, Theta, epsilon, g, kappa, gamma, namespace = 'jc2')

SYS = (JC2 + cid(1)) << P_sigma(0, 2, 1) << (JC1 + cid(1))
```

The resulting system's block diagram is:



and its overall SLH model is given by:

$$\left(\begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0\end{pmatrix}, \begin{pmatrix}\sqrt{\kappa}a_{\mathrm{fock}_{\mathrm{jc1}}} + \sqrt{\kappa}a_{\mathrm{fock}_{\mathrm{jc2}}}\\ \sqrt{\gamma}\sigma_{\mathrm{g,e}}^{\mathrm{tls}_{\mathrm{jc2}}}\\ \sqrt{\gamma}\sigma_{\mathrm{g,e}}^{\mathrm{tls}_{\mathrm{jc1}}}\end{pmatrix}, \Delta\Pi_{\mathrm{e}}^{\mathrm{tls}_{\mathrm{jc1}}} + \Delta\Pi_{\mathrm{e}}^{\mathrm{tls}_{\mathrm{jc2}}} + \imath g\left(a_{\mathrm{fock}_{\mathrm{jc1}}}^{\dagger}\sigma_{\mathrm{g,e}}^{\mathrm{tls}_{\mathrm{jc1}}} - a_{\mathrm{fock}_{\mathrm{jc1}}}\sigma_{\mathrm{e,g}}^{\mathrm{tls}_{\mathrm{jc1}}}\right) + \imath g\left(a_{\mathrm{fock}_{\mathrm{jc2}}}^{\dagger}\sigma_{\mathrm{g,e}}^{\mathrm{tls}_{\mathrm{jc2}}} - a_{\mathrm{fock}_{\mathrm{jc}}}\right.$$

The Super-Operator Algebra module

The specification of a quantum mechanics symbolic super-operator algebra. Each super-operator has an associated *space* property which gives the Hilbert space on which the operators the super-operator acts non-trivially are themselves acting non-trivially.

The most basic way to construct super-operators is by lifting 'normal' operators to linear pre- and post-multiplication super-operators:

```
>>> A, B, C = OperatorSymbol("A", FullSpace), OperatorSymbol("B", FullSpace),␣
↪OperatorSymbol("C", FullSpace)
>>> SPre(A) * B
    A * B
>>> SPost(C) * B
    B * C
>>> (SPre(A) * SPost(C)) * B
    A * B * C
```

```
>>> (SPre(A) - SPost(A)) * B        # Linear super-operator associated with A that␣
↪maps B --> [A,B]
    A * B - B * A
```

There exist some useful constants to specify neutral elements of super-operator addition and multiplication:

`ZeroSuperOperator` `IdentitySuperOperator`

Super operator objects can be added together in code via the infix '+' operator and multiplied with the infix '*' operator. They can also be added to or multiplied by scalar objects. In the first case, the scalar object is multiplied by the IdentitySuperOperator constant.

Super operators are applied to operators by multiplying an operator with superoperator from the left:

```
>>> S = SuperOperatorSymbol("S", FullSpace)
>>> A = OperatorSymbol("A", FullSpace)
>>> S * A
    SuperOperatorTimesOperator(S, A)
>>> isinstance(S*A, Operator)
    True
```

The result is an operator.

The State (Ket-) Algebra module

This module implements a basic Hilbert space state algebra where by default we represent states ψ as 'Ket' vectors $\psi \to |\psi\rangle$. However, any state can also be represented in its adjoint Bra form, since those representations are dual:

$$\psi \leftrightarrow |\psi\rangle \leftrightarrow \langle\psi|$$

States can be added to states of the same Hilbert space. They can be multiplied by:

- scalars, to just yield a rescaled state within the original space
- operators that act on some of the states degrees of freedom (but none that aren't part of the state's Hilbert space)
- other states that have a Hilbert space corresponding to a disjoint set of degrees of freedom

Furthermore,

- a `Ket` object can multiply a `Bra` of the same space from the left to yield a `KetBra` type operator.

And conversely,

- a `Bra` can multiply a `Ket` from the left to create a (partial) inner product object `BraKet`. Currently, only full inner products are supported, i.e. the `Ket` and `Bra` operands need to have the same space.

CHAPTER 3

Properties and Simplification of Circuit Algebraic Expressions

By observing that we can define for a general system $Q = (S, L, H)$ its *series inverse* system $Q^{\lhd -1} := (S^\dagger, -S^\dagger L, -H)$

$$(S, L, H) \lhd (S^\dagger, -S^\dagger L, -H) = (S^\dagger, -S^\dagger L, -H) \lhd (S, L, H) = (\mathbb{I}_n, 0, 0) =: \mathrm{id}_n,$$

we see that the series product induces a group structure on the set of n-channel circuit components for any $n \geq 1$. It can easily be verified that the series inverse of the basic operations is calculated as follows

$$\begin{aligned}(Q_1 \lhd Q_2)^{\lhd -1} &= Q_2^{\lhd -1} \lhd Q_1^{\lhd -1} \\ (Q_1 \boxplus Q_2)^{\lhd -1} &= Q_1^{\lhd -1} \boxplus Q_2^{\lhd -1} \\ ([Q]_{k \to l})^{\lhd -1} &= \left[Q^{\lhd -1}\right]_{l \to k}.\end{aligned}$$

In the following, we denote the number of channels of any given system $Q = (S, L, H)$ by $\mathrm{cdim}\; Q := n$. The most obvious expression simplification is the associative expansion of concatenations and series:

$$\begin{aligned}(A_1 \lhd A_2) \lhd (B_1 \lhd B_2) &= A_1 \lhd A_2 \lhd B_1 \lhd B_2 \\ (C_1 \boxplus C_2) \boxplus (D_1 \boxplus D_2) &= C_1 \boxplus C_2 \boxplus D_1 \boxplus D_2\end{aligned}$$

A further interesting property that follows intuitively from the graphical representation (cf.~Fig.~ref{fig:decomposition_law}) is the following tensor decomposition law

$$(A \boxplus B) \lhd (C \boxplus D) = (A \lhd C) \boxplus (B \lhd D),$$

which is valid for $\mathrm{cdim}\; A = \mathrm{cdim}\; C$ and $\mathrm{cdim}\; B = \mathrm{cdim}\; D$.

The following figures demonstrate the ambiguity of the circuit algebra:

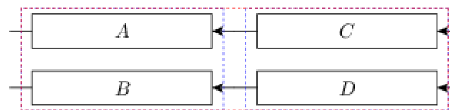


Fig. 3.1: $(A \boxplus B) \lhd (C \boxplus D)$

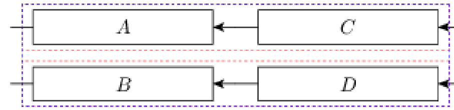


Fig. 3.2: $(A \lhd C) \boxplus (B \lhd D)$

Here, a red box marks a series product and a blue box marks a concatenation. The second version expression has the advantage of making more explicit that the overall circuit consists of two channels without direct optical scattering.

It will most often be preferable to use the RHS expression of the tensor decomposition law above as this enables us to understand the flow of optical signals more easily from the algebraic expression. In *[GoughJames09]* Gough and James denote a system that can be expressed as a concatenation as *reducible*. A system that cannot be further decomposed into concatenated subsystems is accordingly called *irreducible*. As follows intuitively from a graphical representation any given complex system $Q = (S, L, H)$ admits a decomposition into $1 \leq N \leq \mathrm{cdim}\ Q$ irreducible subsystems $Q = Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N$, where their channel dimensions satisfy $\mathrm{cdim}\ Q_j \geq 1,\ j = 1, 2, \ldots N$ and $\sum_{j=1}^{N} \mathrm{cdim}\ Q_j = \mathrm{cdim}\ Q$. While their individual parameter triplets themselves are not uniquely determinedfootnote{Actually the scattering matrices $\{S_j\}$ and the coupling vectors $\{L_j\}$ *are* uniquely determined, but the Hamiltonian parameters $\{H_j\}$ must only obey the constraint $\sum_{j=1}^{N} H_j = H$.}, the sequence of their channel dimensions $(\mathrm{cdim}\ Q_1, \mathrm{cdim}\ Q_2, \ldots \mathrm{cdim}\ Q_N) =: \mathrm{bls}\ Q$ clearly is. We denote this tuple as the block structure of Q. We are now able to generalize the decomposition law in the following way: Given two systems of n channels with the same block structure $\mathrm{bls}\ A = \mathrm{bls}\ B = (n_1, ...n_N)$, there exist decompositions of A and B such that

$$A \lhd B = (A_1 \lhd B_1) \boxplus \cdots \boxplus (A_N \lhd B_N)$$

with $\mathrm{cdim}\ A_j = \mathrm{cdim}\ B_j = n_j,\ j = 1, \ldots N$. However, even in the case that the two block structures are not equal, there may still exist non-trivial compatible block decompositions that at least allow a partial application of the decomposition law. Consider the example presented in Figure (block_structures).

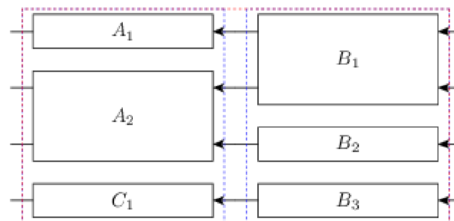


Fig. 3.3: Series "$(1, 2, 1) \lhd (2, 1, 1)$"

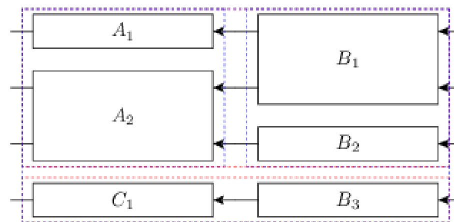


Fig. 3.4: Optimal decomposition into $(3, 1)$

Even in the case of a series between systems with unequal block structures, there often exists a non-trivial common block decomposition that simplifies the overall expression.

Permutation objects

The algebraic representation of complex circuits often requires systems that only permute channels without actual scattering. The group of permutation matrices is simply a subgroup of the unitary (operator) matrices. For any permutation matrix P, the system described by $(P, 0, 0)$ represents a pure permutation of the optical fields (ref fig permutation).

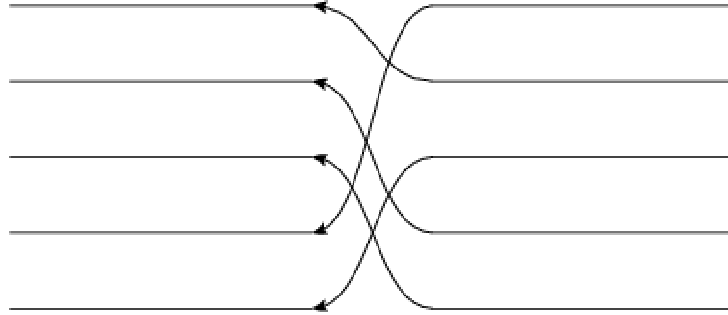


Fig. 3.5: A graphical representation of P_σ where $\sigma \equiv (4, 1, 5, 2, 3)$ in image tuple notation.

A permutation σ of n elements ($\sigma \in \Sigma_n$) is often represented in the following form $\begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix}$, but obviously it is also sufficient to specify the tuple of images $(\sigma(1), \sigma(2), \dots, \sigma(n))$. We now define the permutation matrix via its matrix elements

$$(P_\sigma)_{kl} = \delta_{k\sigma(l)} = \delta_{\sigma^{-1}(k)l}.$$

Such a matrix then maps the j-th unit vector onto the $\sigma(j)$-th unit vector or equivalently the j-th incoming optical channel is mapped to the $\sigma(j)$-th outgoing channel. In contrast to a definition often found in mathematical literature this definition ensures that the representation matrix for a composition of permutations $\sigma_2 \circ \sigma_1$ results from a product of the individual representation matrices in the same order $P_{\sigma_2 \circ \sigma_1} = P_{\sigma_2} P_{\sigma_1}$. This can be shown directly on the order of the matrix elements

$$\begin{aligned}(P_{\sigma_2\circ\sigma_1})_{kl} &= \delta_{k(\sigma_2\circ\sigma_1)(l)} = \sum_j \delta_{kj}\delta_{j(\sigma_2\circ\sigma_1)(l)} = \sum_j \delta_{k\sigma_2(j)}\delta_{\sigma_2(j)(\sigma_2\circ\sigma_1)(l)} \\ &= \sum_j \delta_{k\sigma_2(j)}\delta_{\sigma_2(j)\sigma_2(\sigma_1(l))} = \sum_j \delta_{k\sigma_2(j)}\delta_{j\sigma_1(l)} = \sum_j (P_{\sigma_2})_{kj}(P_{\sigma_1})_{jl},\end{aligned}$$

where the third equality corresponds simply to a reordering of the summands and the fifth equality follows from the bijectivity of σ_2. In the following we will often write P_σ as a shorthand for $(P_\sigma, 0, 0)$. Thus, our definition ensures that we may simplify any series of permutation systems in the most intuitive way: $P_{\sigma_2} \lhd P_{\sigma_1} = P_{\sigma_2\circ\sigma_1}$. Obviously the set of permutation systems of n channels and the series product are a subgroup of the full system series group of n channels. Specifically, it includes the identity $\mathrm{id}n = P_{\sigma_{\mathrm{id}_n}}$.

From the orthogonality of the representation matrices it directly follows that $P_\sigma^T = P_{\sigma^{-1}}$ For future use we also define a concatenation between permutations

$$\sigma_1 \boxplus \sigma_2 := \begin{pmatrix} 1 & 2 & \dots & n & n+1 & n+2 & \dots & n+m \\ \sigma_1(1) & \sigma_1(2) & \dots & \sigma_1(n) & n+\sigma_2(1) & n+\sigma_2(2) & \dots & n+\sigma_2(m) \end{pmatrix},$$

which satisfies $P_{\sigma_1} \boxplus P_{\sigma_2} = P_{\sigma_1 \boxplus \sigma_2}$ by definition. Another helpful definition is to introduce a special set of permutations that map specific ports into each other but leave the relative order of all other ports intact:

$$\omega_{l\leftarrow k}^{(n)} := \begin{cases} \begin{pmatrix} 1 & \dots & k-1 & k & k+1 & \dots & l-1 & l & l+1 & \dots & n \\ 1 & \dots & k-1 & l & k & \dots & l-2 & l-1 & l+1 & \dots & n \end{pmatrix} & \text{for } k < l \\ \begin{pmatrix} 1 & \dots & l-1 & l & l+1 & \dots & k-1 & k & k+1 & \dots & n \\ 1 & \dots & l-1 & l+1 & l+2 & \dots & k & l & k+1 & \dots & n \end{pmatrix} & \text{for } k > l \end{cases}$$

We define the corresponding system objects as $W^{(n)}_{l\leftarrow k} := P_{\omega^{(n)}_{l\leftarrow k}}$.

Permutations and Concatenations

Given a series $P_\sigma \triangleleft (Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N)$ where the Q_j are irreducible systems, we analyze in which cases it is possible to (partially) "move the permutation through" the concatenated expression. Obviously we could just as well investigate the opposite scenario $(Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N) \triangleleft P_\sigma$, but this second scenario is closely relatedfootnote{Series-Inverting a series product expression also results in an inverted order of the operand inverses $(Q_1 \triangleleft Q_2)^{\triangleleft -1} = Q_2^{\triangleleft -1} \triangleleft Q_1^{\triangleleft -1}$. Since the inverse of a permutation (concatenation) is again a permutation (concatenation), the cases are in a way "dual" to each other.}.

Block-permuting permutations

The simples case is realized when the permutation simply permutes whole blocks intactly

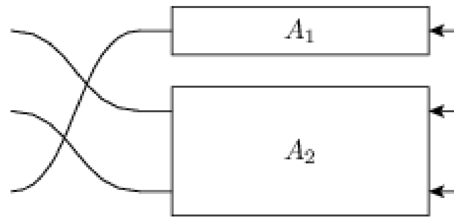


Fig. 3.6: $P_\sigma \triangleleft (A_1 \boxplus A_2)$

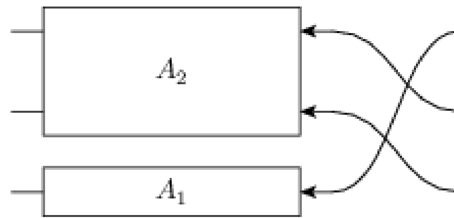


Fig. 3.7: $(A_2 \boxplus A_1) \triangleleft P_\sigma$

A block permuting series.

Given a block structure $n := (n_1, n_2, \ldots n_N)$ a permutation $\sigma \in \Sigma_n$ is said to *block permute* n iff there exists a permutation $\tilde{\sigma} \in \Sigma_N$ such that

$$\begin{aligned} P_\sigma \triangleleft (Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N) &= (P_\sigma \triangleleft (Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N) \triangleleft P_{\sigma^{-1}}) \triangleleft P_\sigma \\ &= (Q_{\tilde{\sigma}(1)} \boxplus Q_{\tilde{\sigma}(2)} \boxplus \cdots \boxplus Q_{\tilde{\sigma}(N)}) \triangleleft P_\sigma \end{aligned}$$

Hence, the permutation σ, given in image tuple notation, block permutes n iff for all $1 \leq j \leq N$ and for all $0 \leq k < n_j$ we have $\sigma(o_j + k) = \sigma(o_j) + k$, where we have introduced the block offsets $o_j := 1 + \sum_{j' < j} n_j$. When these conditions are satisfied, $\tilde{\sigma}$ may be obtained by demanding that $\tilde{\sigma}(a) > \tilde{\sigma}(b) \Leftrightarrow \sigma(o_a) > \sigma(o_b)$. This equivalence reduces the computation of $\tilde{\sigma}$ to sorting a list in a specific way.

Block-factorizing permutations

The next-to-simplest case is realized when a permutation σ can be decomposed $\sigma = \sigma_{\rm b} \circ \sigma_{\rm i}$ into a permutation $\sigma_{\rm b}$ that block permutes the block structure n and an internal permutation $\sigma_{\rm i}$ that only permutes within each block,

i.e.~:math:*sigma_{rm i} = sigma_1 boxplus sigma_2 boxplus dots boxplus sigma_N*. In this case we can perform the following simplifications

$$P_\sigma \lhd (Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N) = P_{\sigma_b} \lhd [(P_{\sigma_1} \lhd Q_1) \boxplus (P_{\sigma_2} \lhd Q_2) \boxplus \cdots \boxplus (P_{\sigma_N} \lhd Q_N)] .$$

We see that we have reduced the problem to the above discussed case. The result is now

$$P_\sigma \lhd (Q_1 \boxplus \cdots \boxplus Q_N) = \left[(P_{\sigma_{\tilde{\sigma_b}(1)}} \lhd Q_{\tilde{\sigma_b}(1)}) \boxplus \cdots \boxplus (P_{\sigma_{\tilde{\sigma_b}(N)}} \lhd Q_{\tilde{\sigma_b}(N)})\right] \lhd P_{\sigma_b}.$$

In this case we say that σ *block factorizes* according to the block structure n. The following figure illustrates an example of this case.

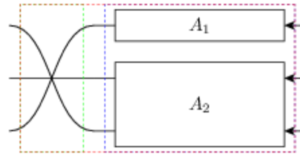


Fig. 3.8: $P_\sigma \lhd (A_1 \boxplus A_2)$

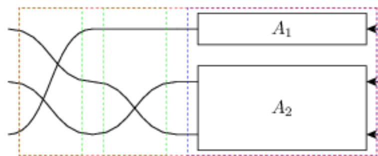


Fig. 3.9: $P_{\sigma_b} \lhd P_{\sigma_i} \lhd (A_1 \boxplus A_2)$

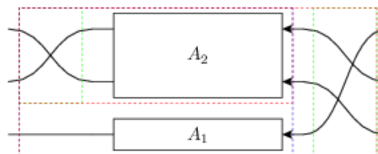


Fig. 3.10: $((P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_b}$

A block factorizable series.

A permutation σ block factorizes according to the block structure n iff for all $1 \le j \le N$ we have $\max_{0 \le k < n_j} \sigma(o_j + k) - \min_{0 \le k' < n_j} \sigma(o_j + k') = n_j - 1$, with the block offsets defined as above. In other words, the image of a single block is coherent in the sense that no other numbers from outside the block are mapped into the integer range spanned by the minimal and maximal points in the block's image. The equivalence follows from our previous result and the bijectivity of σ.

The general case

In general there exists no unique way how to split apart the action of a permutation on a block structure. However, it is possible to define a some rules that allow us to "move as much of the permutation" as possible to the RHS of the series. This involves the factorization $\sigma = \sigma_x \circ \sigma_b \circ \sigma_i$ defining a specific way of constructing both σ_b and σ_i from σ. The remainder σ_x can then be calculated through

$$\sigma_x := \sigma \circ \sigma_i^{-1} \circ \sigma_b^{-1}.$$

Hence, by construction, $\sigma_b \circ \sigma_i$ factorizes according to n so only σ_x remains on the exterior LHS of the expression.

So what then are the rules according to which we construct the block permuting $\sigma_{\rm b}$ and the decomposable $\sigma_{\rm i}$? We wish to define $\sigma_{\rm i}$ such that the remainder $\sigma \circ \sigma_{\rm i}^{-1} = \sigma_{\rm x} \circ \sigma_{\rm b}$ does not cross any two signals that are emitted from the same block. Since by construction $\sigma_{\rm b}$ only permutes full blocks anyway this means that $\sigma_{\rm x}$ also does not cross any two signals emitted from the same block. This completely determines $\sigma_{\rm i}$ and we can therefore calculate $\sigma \circ \sigma_{\rm i}^{-1} = \sigma_{\rm x} \circ \sigma_{\rm b}$ as well. To construct $\sigma_{\rm b}$ it is sufficient to define an total order relation on the blocks that only depends on the block structure n and on $\sigma \circ \sigma_{\rm i}^{-1}$. We define the order on the blocks such that they are ordered according to their minimal image point under σ. Since $\sigma \circ \sigma_{\rm i}^{-1}$ does not let any block-internal lines cross, we can thus order the blocks according to the order of the images of the first signal $\sigma \circ \sigma_{\rm i}^{-1}(o_j)$. In (ref fig general_factorization) we have illustrated this with an example.

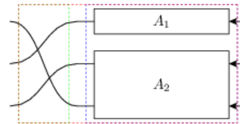


Fig. 3.11: $P_\sigma \lhd (A_1 \boxplus A_2)$

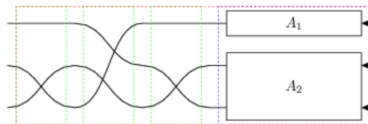


Fig. 3.12: $P_{\sigma_{\rm x}} \lhd P_{\sigma_{\rm b}} \lhd P_{\sigma_{\rm i}} \lhd (A_1 \boxplus A_2)$

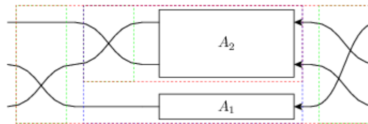


Fig. 3.13: $(P_{\sigma_{\rm x}} \lhd (P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_{\rm b}}$

A general series with a non-factorizable permutation. In the intermediate step we have explicitly separated $\sigma = \sigma_{\rm x} \circ \sigma_{\rm b} \circ \sigma_{\rm i}$.

Finally, it is a whole different question, why we would want move part of a permutation through the concatenated expression in this first place as the expressions usually appear to become more complicated rather than simpler. This is, because we are currently focussing only on single series products between two systems. In a realistic case we have many systems in series and among these there might be quite a few permutations. Here, it would seem advantageous to reduce the total number of permutations within the series by consolidating them where possible: $P_{\sigma_2} \lhd P_{\sigma_1} = P_{\sigma_2 \circ \sigma_1}$. To do this, however, we need to try to move the permutations through the full series and collect them on one side (in our case the RHS) where they can be combined to a single permutation. Since it is not always possible to move a permutation through a concatenation (as we have seen above), it makes sense to at some point in the simplification process reverse the direction in which we move the permutations and instead collect them on the LHS. Together these two strategies achieve a near perfect permutation simplification.

Feedback of a concatenation

A feedback operation on a concatenation can always be simplified in one of two ways: If the outgoing and incoming feedback ports belong to the same irreducible subblock of the concatenation, then the feedback can be directly applied only to that single block. For an illustrative example see the figures below:

Reduction to feedback of subblock.

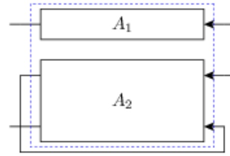


Fig. 3.14: $[A_1 \boxplus A_2]_{2\to 3}$

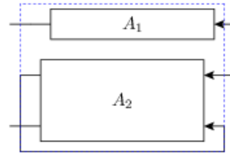


Fig. 3.15: $A_1 \boxplus [A_2]_{1\to 2}$

If, on the other, the outgoing feedback port is on a different subblock than the incoming, the resulting circuit actually does not contain any real feedback and we can find a way to reexpress it algebraically by means of a series product.

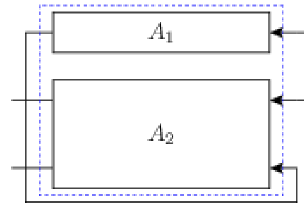


Fig. 3.16: $[A_1 \boxplus A_2]_{1\to 3}$

Reduction of feedback to series, first example

Reduction of feedback to series, second example

To discuss the case in full generality consider the feedback expression $[A \boxplus B]_{k\to l}$ with $\text{cdim } A = n_A$ and $\text{cdim } B = n_B$ and where A and B are not necessarily irreducible. There are four different cases to consider.

- $k, l \leq n_A$: In this case the simplified expression should be $[A]_{k\to l} \boxplus B$
- $k, l > n_A$: Similarly as before but now the feedback is restricted to the second operand $A \boxplus [B]_{(k-n_A)\to(l-n_A)}$, cf. Fig. (ref fig fc_irr).
- $k \leq n_A < l$: This corresponds to a situation that is actually a series and can be re-expressed as $(\text{id}n_A - 1 \boxplus B) \lhd W^{(n)}_{(l-1)\leftarrow k} \lhd (A + \text{id}n_B - 1)$, cf. Fig. (ref fig fc_re1).
- $l \leq n_A < k$: Again, this corresponds a series but with a reversed order compared to above $(A + \text{id}n_B - 1) \lhd W^{(n)}_{l\leftarrow(k-1)} \lhd (\text{id}n_A - 1 \boxplus B)$, cf. Fig. (ref fig fc_re2).

Feedback of a series

There are two important cases to consider for the kind of expression at either end of the series: A series starting or ending with a permutation system or a series starting or ending with a concatenation.

Reduction of series feedback with a concatenation at the RHS

Reduction of series feedback with a permutation at the RHS

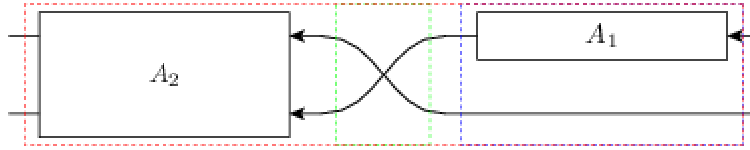


Fig. 3.17: $A_2 \lhd W^{(2)}_{2\leftarrow 1} \lhd (A_2 \boxplus \mathrm{id}_1)$

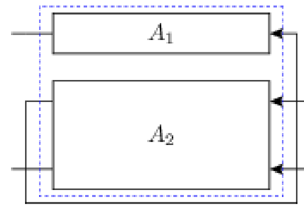


Fig. 3.18: $[A_1 \boxplus A_2]_{2\to 1}$

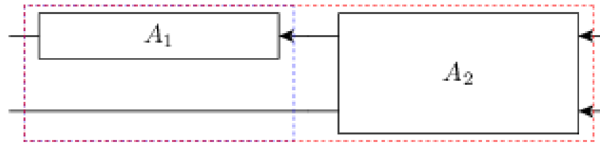


Fig. 3.19: $(A_1 \boxplus \mathrm{id}_1) \lhd A_2$

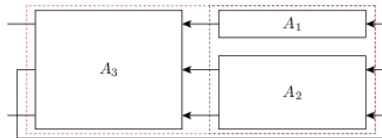


Fig. 3.20: $[A_3 \lhd (A_1 \boxplus A_2)]_{2\to 1}$

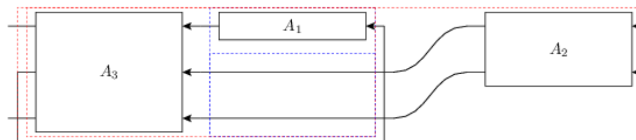


Fig. 3.21: $(A_3 \lhd (A_1 \boxplus \mathrm{id}_2)) \lhd A_2$

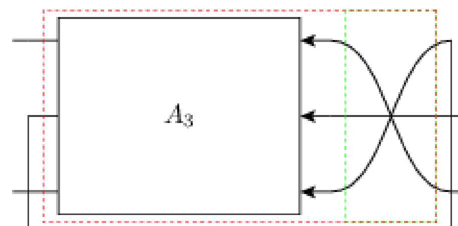


Fig. 3.22: $[A_3 \lhd P_\sigma]_{2\to 1}$

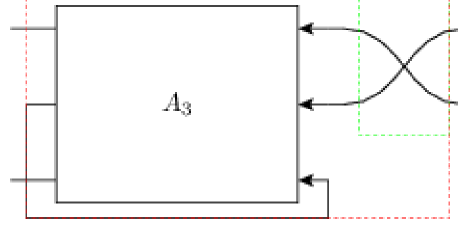


Fig. 3.23: $[A_3]_{2\to 3} \lhd P_{\tilde\sigma}$

1) $[A \lhd (C \boxplus D)]_{k\to l}$: We define $n_C = \text{cdim}\, C$ and $n_A = \text{cdim}\, A$. Without too much loss of generality, let's assume that $l \le n_C$ (the other case is quite similar). We can then pull D out of the feedback loop: $[A \lhd (C \boxplus D)]_{k\to l} \longrightarrow [A \lhd (C \boxplus \text{id}n_D)]_{k\to l} \lhd (\text{id}n_C - 1 \boxplus D)$. Obviously, this operation only makes sense if $D \neq \text{id}n_D$. The case $l > n_C$ is quite similar, except that we pull C out of the feedback. See Figure (ref fig fs_c) for an example.

2. We now consider $[(C \boxplus D) \lhd E]_{k\to l}$ and we assume $k \le n_C$ analogous to above. Provided that $D \neq \text{id}n_D$, we can pull it out of the feedback and get $(\text{id}n_C - 1 \boxplus D) \lhd [(C \boxplus \text{id}n_D) \lhd E]_{k\to l}$.

3) $[A \lhd P_\sigma]_{k\to l}$: The case of a permutation within a feedback loop is a lot more intuitive to understand graphically (e.g., cf. Figure ref fig fs_p). Here, however we give a thorough derivation of how a permutation can be reduced to one involving one less channel and moved outside of the feedback. First, consider the equality $[A \lhd W_{j\leftarrow l}^{(n)}]_{k\to l} = [A]_{k\to j}$ which follows from the fact that $W_{j\leftarrow l}^{(n)}$ preserves the order of all incoming signals except the l-th. Now, rewrite

$$\begin{aligned}
[A \lhd P_\sigma]_{k\to l} &= [A \lhd P_\sigma \lhd W_{l\leftarrow n}^{(n)} \lhd W_{n\leftarrow l}^{(n)}]_{k\to l} \\
&= [A \lhd P_\sigma \lhd W_{l\leftarrow n}^{(n)}]_{k\to n} \\
&= [A \lhd W_{\sigma(l)\leftarrow n}^{(n)} \lhd (W_{n\leftarrow\sigma(l)}^{(n)} \lhd P_\sigma \lhd W_{l\leftarrow n})]_{k\to n}
\end{aligned}$$

Turning our attention to the bracketed expression within the feedback, we clearly see that it must be a permutation system $P_{\sigma'} = W_{n\leftarrow\sigma(l)}^{(n)} \lhd P_\sigma \lhd W_{l\leftarrow n}^{(n)}$ that maps $n \to l \to \sigma(l) \to n$. We can therefore write $\sigma' = \tilde\sigma \boxplus \sigma_{\text{id}_1}$ or equivalently $P_{\sigma'} = P_{\tilde\sigma} \boxplus \text{id1}$ But this means, that the series within the feedback ends with a concatenation and from our above rules we know how to handle this:

$$\begin{aligned}
[A \lhd P_\sigma]_{k\to l} &= [A \lhd W_{\sigma(l)\leftarrow n}^{(n)} \lhd (P_{\tilde\sigma} \boxplus \text{id1})]_{k\to n} \\
&= [A \lhd W_{\sigma(l)\leftarrow n}^{(n)}]_{k\to n} \lhd P_{\tilde\sigma} \\
&= [A]_{k\to\sigma(l)} \lhd P_{\tilde\sigma},
\end{aligned}$$

where we know that the reduced permutation is the well-defined restriction to $n - 1$ elements of $\sigma' = \left(\omega_{n\leftarrow\sigma l}^{(n)} \circ \sigma \circ \omega_{l\leftarrow n}^{(n)}\right)$.

4. The last case is analogous to the previous one and we will only state the results without a derivation:

$$[P_\sigma \lhd A]_{k\to l} = P_{\tilde\sigma} \lhd [A]_{\sigma^{-1}(k)\to l},$$

where the reduced permutation is given by the (again well-defined) restriction of $\omega_{n\leftarrow k}^{(n)} \circ \sigma \circ \omega_{\sigma^{-1}(k)\leftarrow n}^{(n)}$ to $n - 1$ elements.

CHAPTER 4

Circuit Component Definition

The best way to get started on defining one's own circuit component definition is to look at the examples provided in the component library *qnet.circuit_components*. Every circuit component object is a python class definition that derives off the class *qnet.circuit_components.component.Component*. The subclass must necessarily overwrite the following class attributes of this `Component` class:

- `CDIM` needs to be set to the full number (`int`) of input or output noises, i.e., the row dimension of the coupling vector $\mathbf{L}$ or the scattering matrix $\mathbf{S}$ of the corresponding $(\mathbf{S}, \mathbf{L}, H)$ model.
- `PORTSIN` needs to be set to a list of port labels for the relevant input ports of the component, i.e., those that could be connected to other components. The number of entries can be smaller or equal than `CDIM`.
- `PORTSOUT` needs to be set to a list of port labels for the relevant output ports of the component, i.e., those that could be connected to other components. The number of entries can be smaller or equal than `CDIM`.
- If your model depends on parameters you should specify this both via the `_params` attribute and by adding a class attribute with the name of the parameter and a default value that is either numeric or symbolic. Checkout some of the existing modules such as *qnet.circuit_components.single_sided_opo_cc* to see how these parameters should be set.
- If your model has internal quantum degrees of freedom, you need to implement the `_space` property. If your model has a single quantum degree of freedom such as an empty cavity or an OPO, just follow the example of *qnet.circuit_components.single_sided_opo_cc* (click on 'source' to see the source-code). If your model's space will be a tensor product of several degrees of freedom, follow the example of *qnet.circuit_components.single_sided_jaynes_cummings_cc*, which defines Hilbert space properties for the different degrees of freedom and has the `_space` property return a tensor product of them.

  In general, it is important to properly assign a unique name and namespace to all internal degrees of freedom to rule out ambiguities when your final circuit includes more than one instance of your model.
- Optionally, you may overwrite the `name` attribute to change the default name of your component.

Most importantly, the subclass must implement a `_toSLH(self):` method. Doing this requires some knowledge of how to use the operator algebra *qnet.algebra.operator_algebra*. For a component model with multiple input/output ports **with no direct scattering between some ports**, i.e., the scattering matrix $\mathbf{S}$ is (block-) diagonal we allow for a formalism to define this substructure on the circuit-symbolic level by not just defining a component model,

but also models for the irreducible subblocks of your component. This leads to two alternative ways of defining the circuit components:

1. Simple case, creating a symbolically *irreducible* circuit model, this is probably what you should go with:

   This suffices if the purpose of defining the component is only to derive the final quantum equations of motion for an overall system, i.e., no analysis should be carried out on the level of the circuit algebra but only on the level of the underlying operator algebra of the full circuit's $(\mathbf{S}, \mathbf{L}, H)$ model.

   Subclassing the `Component` class takes care of implementing the class constructor `__init__` and this should not be overwritten unless you are sure what you are doing. The pre-defined constructor takes care of handling the flexible specification of model parameters as well as the name and namespace via its arguments. I.e., for a model named `MyModel` whose `_parameters` attribute is given by `['kappa', 'gamma']`, one can either specify all or just some of the parameters as named arguments. The rest get replaced by the default values. Consider the following code examples:

```
MyModel(name = "M")
# -> MyModel(name = "M", namespace = "", kappa = MyModel.kappa, gamma = MyModel.
↪gamma)

MyModel(name = "M", kappa = 1)
# -> MyModel(name = "M", namespace = "", kappa = 1, gamma = MyModel.gamma)

MyModel(kappa = 1)
# -> MyModel(name = MyModel.name, namespace = "", kappa = 1, gamma = MyModel.
↪gamma)
```

   The model parameters passed to the constructor are subsequently accessible to the object's methods as instance attributes. I.e., within the `_toSLH(self)`-method of the above example one would access the value of the `kappa` parameter as `self.kappa`.

2. Complex case, create a symbolically *reducible* circuit model:

   In this case you will need to define subcomponent model for each irreducible block of your model. We will not discuss this advanced method here, but instead refer to the following modules as examples:

   - `qnet.circuit_components.relay_cc`
   - `qnet.circuit_components.single_sided_jaynes_cummings_cc`
   - `qnet.circuit_components.double_sided_opo_cc`

A simple example

As an example we will now define a simple (symbolically irreducible) version of the single sided jaynes cummings model. The model is given by:

$$S = \mathbf{1}_2$$
$$L = \begin{pmatrix} \sqrt{\kappa}a \\ \sqrt{\gamma}\sigma_- \end{pmatrix}$$
$$H = \Delta_f a^\dagger a + \Delta_a \sigma_+ \sigma_- + ig\left(\sigma_+ a - \sigma_- a^\dagger\right)$$

Then, we can define the corresponding component class as:

```
from sympy import symbols, I, sqrt
from qnet.algebra.circuit_algebra import Create, LocalSigma, SLH, Destroy, local_
↪space, Matrix, identity_matrix
```

```
class SingleSidedJaynesCummings(Component):

    CDIM = 2

    name = "Q"

    kappa = symbols('kappa', real = True)       # decay of cavity mode through cavity␣
↪mirror
    gamma = symbols('gamma', real = True)       # decay rate into transverse modes
    g = symbols('g', real = True)               # coupling between cavity mode and␣
↪two-level-system
    Delta_a = symbols('Delta_a', real = True)   # detuning between the external␣
↪driving field and the atom
    Delta_f = symbols('Delta_f', real = True)   # detuning between the external␣
↪driving field and the cavity
    FOCK_DIM = 20                               # default truncated Fock-space␣
↪dimension

    _parameters = ['kappa', 'gamma', 'g', 'Delta_a', 'Delta_f', 'FOCK_DIM']

    PORTSIN = ['In1', 'VacIn']
    PORTSOUT = ['Out1', 'UOut']

    @property
    def fock_space(self):
        """The cavity mode's Hilbert space."""
        return local_space("f", make_namespace_string(self.namespace, self.name),␣
↪dimension = self.FOCK_DIM)

    @property
    def tls_space(self):
        """The two-level-atom's Hilbert space."""
        return local_space("a", make_namespace_string(self.namespace, self.name),␣
↪basis = ('h', 'g'))

    @property
    def _space(self):
        return self.fock_space * self.tls_space


    def _toSLH(self):
        a = Destroy(self.fock_space)
        sigma = LocalSigma(self.tls_space, 'g', 'h')
        H = self.Delta_f * a.dag() * a + self.Delta_a * sigma.dag() * sigma \
                + I * self.g * (sigma.dag() * a - sigma * a.dag())
        L1 = sqrt(self.kappa) * a
        L2 = sqrt(self.gamma) * sigma
        L = Matrix([[L1],
                    [L2]])
        S = identity_matrix(2)
        return SLH(S, L, H)
```

Creating custom component symbols for `gschem`

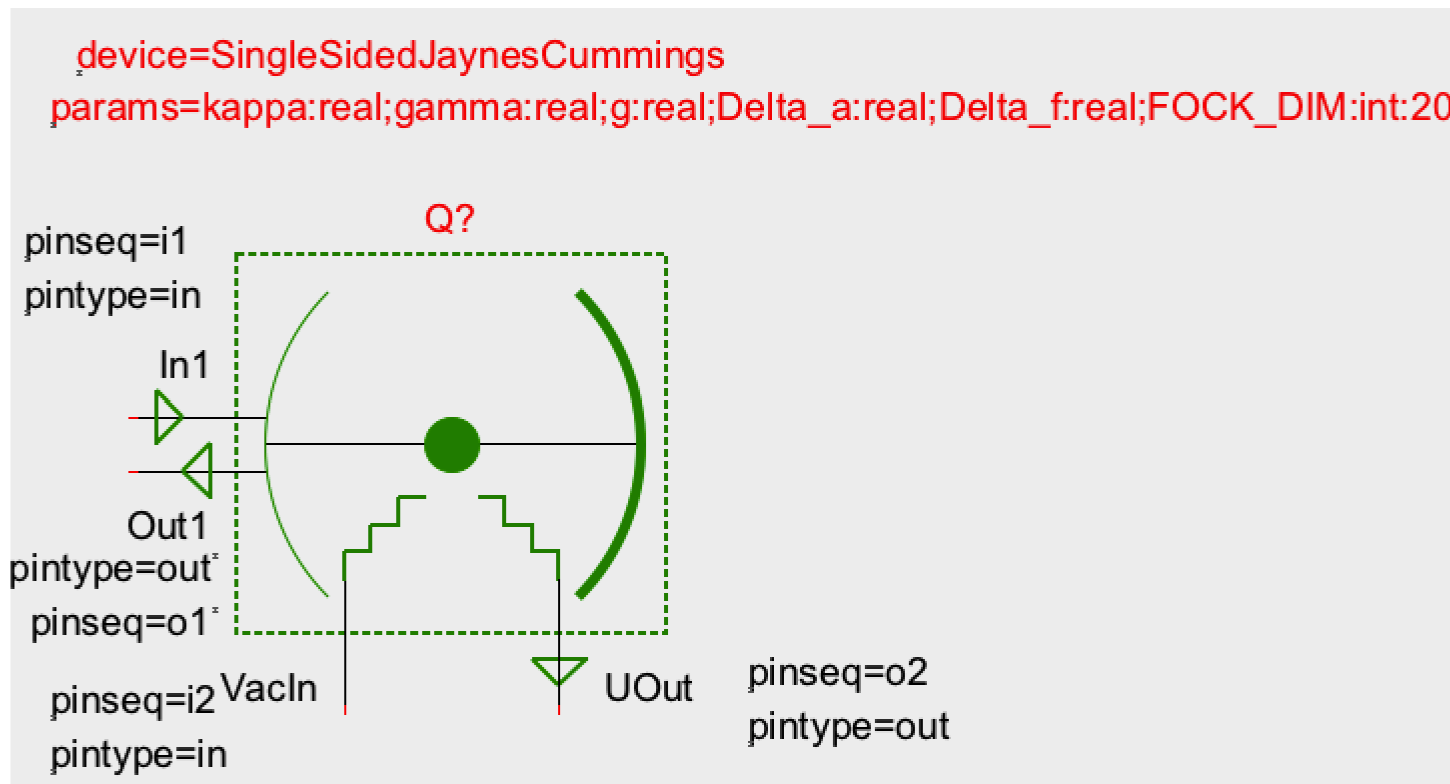
Creating symbols in gschem is similar to the schematic capture process itself:

1. Using the different graphical objects (lines, boxes, arcs, text) create the symbol as you see fit.
2. Add pins for the symbols inputs and outputs. Define their `pintype` (`in` or `out`) and their `pinnumber` (which can be text or a number) according to the port names. Finally, define their `pinseq` attributes to match the order of the list in the python component definition, so for the above example, one would need 4 pins, two inputs, two outputs with the following properties:
   - `pintype=in, pinnumber=In1, pinseq=i1`
   - `pintype=in, pinnumber=VacIn, pinseq=i2`
   - `pintype=out, pinnumber=Out1, pinseq=o1`
   - `pintype=out, pinnumber=UOut, pinseq=o2`
3. Define the parameters the model depends on, by adding a `params` attribute to the top level circuit. For the example above the correct param string would be:

```
kappa:real;gamma:real;g:real;Delta_a:real;Delta_f:real;FOCK_DIM:int:20
```

4. Add the name of the component by setting the `device` top-level-attribute, in this example to `SingleSidedJaynesCummings`
5. Specify the default name by adding a `refdes` attribute that is equal to the default name plus an appended question mark (e.g. `Q?`). When designing a circuit, this helps to quickly identify unnamed subcomponents.

The result could look something like this:



CHAPTER 5

Schematic Capture

Here we explain how to create photonic circuits visually using `gschem`

1. From the 'Add' menu select 'Component' to open the component symbol library.
2. Layout components on the grid
3. Double-click the component symbols to edit the properties of each component instance. Be sure to set a unique instance identifier `refdes`-attribute.

   If a component symbol has an attribute named `params`, its value should be understood as a list of the form: `param1_name:param1_type[:default_value1]; param2_name:param2_type[:default_value2];...` where the default values are optional. To assign a value to a component param, add an attribute of the param name and set the value either to a corresponding numerical value or to a parameter name of the overall circuit.
4. For all input and output ports of the circuit that are part of its external interface add dedicated input and output pad objects. Assign names to them (`refdes`-attribute) that correspond to their port names and assign sequence numbers to them, numbering the inputs as `i1, i2, ...` and the outputs as `o1, o2, ...`
5. Draw all internal signals to connect component ports with each other and with port objects.
6. Add a `params`-attribute to the whole circuit specifying all model parameters similarly to above.
7. Add a `module-name`-attribute to the whole circuit to specify its entity name. Please use `CamelCaseConventions` for naming your circuit, because it will ultimately be the name of a Python class.

As an example, consider this screencast for creating a Pseudo-NAND-Latch.

CHAPTER 6

Netlisting

Using gnetlist

Given a well-formed gschem circuit specification file we can use the gnetlist tool that comes with the gEDA suite to export it to a QHDL-file.

Using the command-line, if the .sch schematic file is located at the path my_dir/my_schematic.sch, and you wish to produce a QHDL file at the location my_other_dir/my_netlist.qhdl, run the following command:

```
gnetlist -g qhdl my_dir/my_schematic.sch -o my_other_dir/my_netlist.qhdl
```

It is generally a very good idea to inspect the produced QHDL file code and verify that it looks like it should before trying to compile it into a python circuit_component library file.

The QHDL Syntax

A QHDL file consists of two basic parts:

1. An entity declaration, which should be thought of as defining the external interface of the specified circuit. I.e., it defines global input and output ports as well as parameters for the overall model.
2. A corresponding architecture declaration, that, in turn consists of two parts:
   (a) The architecture head defines what *types* of components can appear in the circuit. I.e., for each component declaration in the architecture head, there can exist multiple *instances* of that component type in the circuit. The head also defines the internal signal lines of the circuit.
   (b) The architecture body declares what instances of which component type exists in the circuit, how its ports are mapped to the internal signals or entity ports, and how its internal parameters relate to the entity parameters. In QHDL, each signal may only connect exactly two ports, where one of three cases is true:
      i. It connects an entity input with a component instance input
      ii. It connects an entity output with a component instance output

iii. It connects a component output with a component input

Before showing some examples of QHDL files, we present the general QHDL syntax in somewhat abstract form. Here, square brackets `[optional]` denote optional keywords/syntax and the ellipses `...` denote repetition:

```
-- this is a comment

-- entity definition
-- this serves as the external interface to the circuit, specifying inputs and outputs
-- as well as parameters of the model
entity my_entity is
    [generic ( var1: generic_type [:= default_var1]] [; var2: generic_type [...] ...
↪]);]
    port (i_1,i_2,...i_n:in fieldmode; o_1,o_2,...o_n:out fieldmode);
end entity my_entity;

-- architecture definition
-- this is the actual implementation of the entity in terms of subcomponents
architecture my_architecture of my_entity is
    -- architecture head
    -- each type of subcomponent, i.e. its ports and its parameters are defined here␣
↪similarly
    -- to the entity definition above
    component my_component
        [generic ( var3: generic_type [:= default_var3]] [; var4: generic_type [...] .
↪..]);]
        port (p1,p2,...pm:in fieldmode; q1,q2,...qm:out fieldmode);
    end component my_component;

    [component my_second_component
        [generic ( var5: generic_type [:= default_var5]] [; var6: generic_type [...] .
↪..]);]
        port (p1,p2,...pr:in fieldmode; q1,q2,...qr:out fieldmode);

    end component my_second_component;

    ...

    ]

    -- internal signals to connect component instances
    [signal s_1,s_2,s_3,...s_m fieldmode;]



begin
    -- architecture body
    -- here the actual component instances are defined and their ports are mapped to␣
↪signals
    -- or to global (i.e. entity-) ports
    -- furthermore, global (entity-) parameters are mapped to component instance␣
↪parameters.

    COMPONENT_INSTANCE_ID1: my_component
        [generic map(var1 => var3, var1 => var4);]
        port map (i_1, i_2, ... i_m, s_1, s_2, ...s_m);

    [COMPONENT_INSTANCE_ID2: my_component
        [generic map(var1 => var3, var1 => var4);]
```

```
        port map (s_1, s_2, ... s_m, o_1, o_2, ...o_m);

    COMPONENT_INSTANCE_ID3: my_second_component
        [generic map (...);]
        port map (...);
    ...
        ]

end architecture my_architecture;
```

where `generic_type` is one of `int`, `real`, or `complex`.

QHDL-Example files:

A Mach-Zehnder-circuit

This toy-circuit realizes a Mach-Zehnder interferometer.

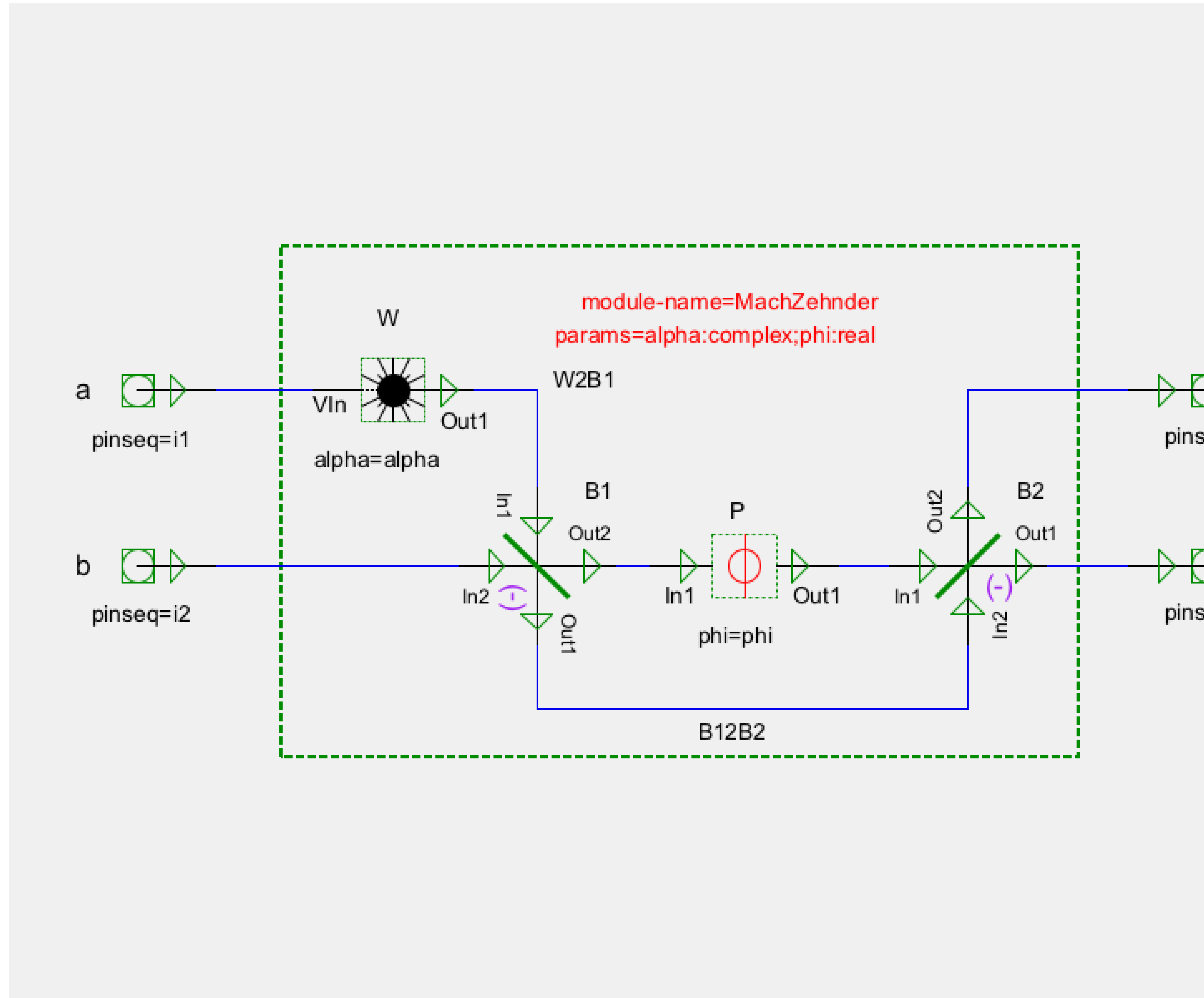
```
-- Structural QHDL generated by gnetlist
-- Entity declaration

ENTITY MachZehnder IS
    GENERIC (
        alpha : complex;
        phi : real);
    PORT (
        a : in fieldmode;
        b : in fieldmode;
        c : out fieldmode;
        d : out fieldmode);
END MachZehnder;


-- Secondary unit
ARCHITECTURE netlist OF MachZehnder IS
    COMPONENT Phase
    GENERIC (
        phi : real);
    PORT (
        In1 : in fieldmode;
        Out1 : out fieldmode);
    END COMPONENT ;

    COMPONENT Beamsplitter
    GENERIC (
        theta : real := 0.7853981633974483);
    PORT (
        In1 : in fieldmode;
        In2 : in fieldmode;
        Out1 : out fieldmode;
        Out2 : out fieldmode);
    END COMPONENT ;

    COMPONENT Displace
    GENERIC (
        alpha : complex);
```


module-name=MachZehnder
params=alpha:complex;phi:real
W
a
pinseq=i1
VIn
Out1
alpha=alpha
W2B1
b
pinseq=i2
In1
B1
Out2
In2
(-)
Out1
P
In1
Out1
phi=phi
Out2
B2
Out1
In1
(-)
In2
B12B2

```
    PORT (
        VIn : in fieldmode;
        Out1 : out fieldmode);
    END COMPONENT ;

    SIGNAL B12B2 : fieldmode;
    SIGNAL W2B1 : fieldmode;
    SIGNAL P2B2 : fieldmode;
    SIGNAL B12P : fieldmode;
    SIGNAL unnamed_net4 : fieldmode;
    SIGNAL unnamed_net3 : fieldmode;
    SIGNAL unnamed_net2 : fieldmode;
    SIGNAL unnamed_net1 : fieldmode;
BEGIN
-- Architecture statement part
    W : Displace
    GENERIC MAP (
        alpha => alpha);
    PORT MAP (
        VIn => unnamed_net1,
        Out1 => W2B1);

    B2 : Beamsplitter
    PORT MAP (
        In1 => P2B2,
        In2 => B12B2,
        Out1 => unnamed_net4,
        Out2 => unnamed_net3);

    B1 : Beamsplitter
    PORT MAP (
        In1 => W2B1,
        In2 => unnamed_net2,
        Out1 => B12B2,
        Out2 => B12P);

    P : Phase
    GENERIC MAP (
        phi => phi);
    PORT MAP (
        In1 => B12P,
        Out1 => P2B2);

-- Signal assignment part
unnamed_net2 <= b;
unnamed_net1 <= a;
d <= unnamed_net4;
c <= unnamed_net3;
END netlist;
```

A Pseudo-NAND-gate

This circuit consists of a Kerr-nonlinear cavity, a few beamsplitters and a bias input amplitude to realize a NAND-gate for the inputs A and B. For details see *[Mabuchi11]*.

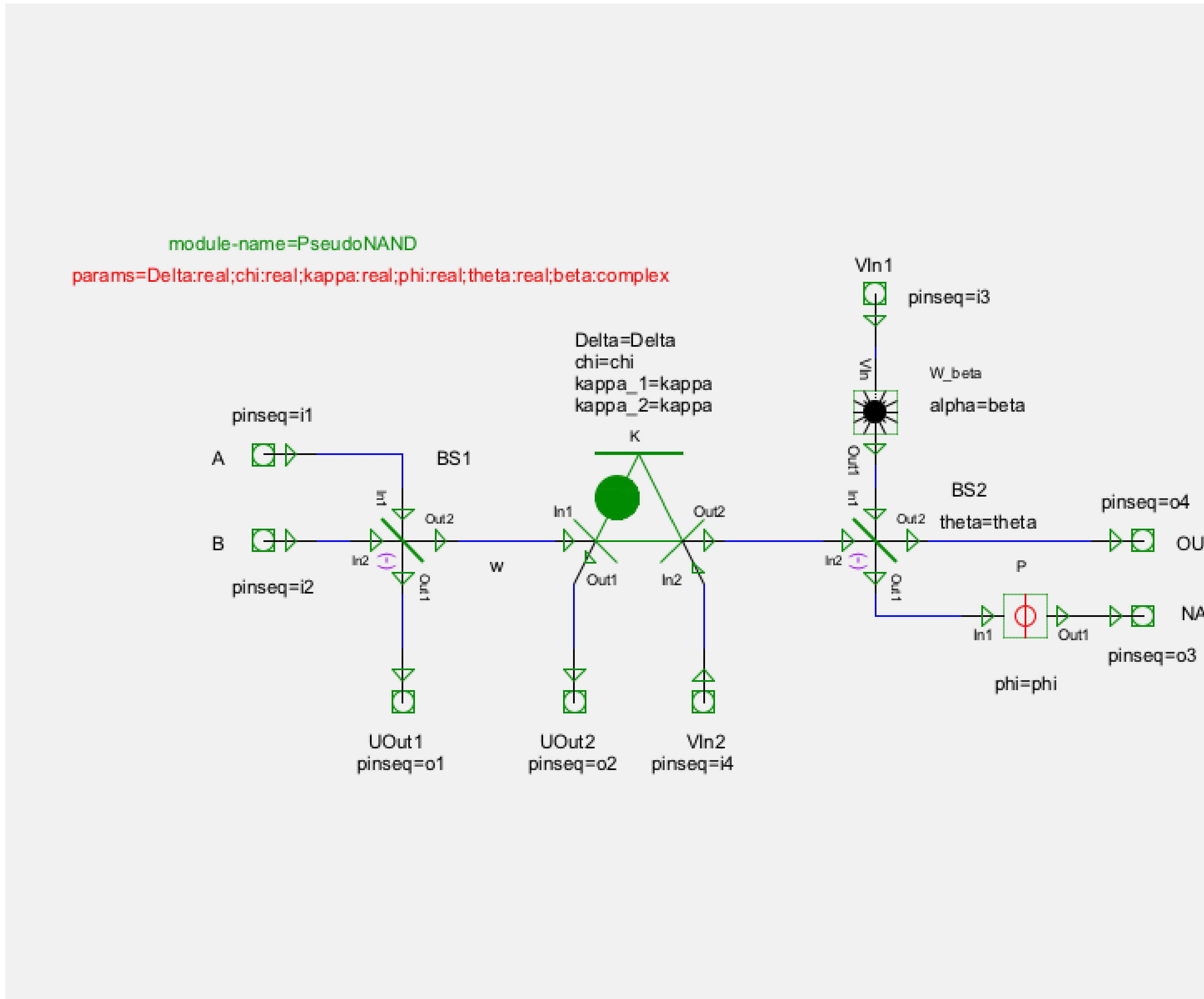


Fig. 6.1: The `gschem` schematic from which the QHDL file below was automatically created.

```
-- Structural QHDL generated by gnetlist
-- Entity declaration

ENTITY PseudoNAND IS
    GENERIC (
        Delta : real;
        chi : real;
        kappa : real;
        phi : real;
        theta : real;
        beta : complex);
    PORT (
        A : in fieldmode;
        B : in fieldmode;
        VIn1 : in fieldmode;
        VIn2 : in fieldmode;
        UOut1 : out fieldmode;
        UOut2 : out fieldmode;
        NAND_AB : out fieldmode;
        OUT2 : out fieldmode);
END PseudoNAND;


-- Secondary unit
ARCHITECTURE netlist OF PseudoNAND IS
    COMPONENT KerrCavity
    GENERIC (
        Delta : real;
        chi : real;
        kappa_1 : real;
        kappa_2 : real);
    PORT (
        In1 : in fieldmode;
        In2 : in fieldmode;
        Out1 : out fieldmode;
        Out2 : out fieldmode);
    END COMPONENT ;

    COMPONENT Phase
    GENERIC (
        phi : real);
    PORT (
        In1 : in fieldmode;
        Out1 : out fieldmode);
    END COMPONENT ;

    COMPONENT Beamsplitter
    GENERIC (
        theta : real := 0.7853981633974483);
    PORT (
        In1 : in fieldmode;
        In2 : in fieldmode;
        Out1 : out fieldmode;
        Out2 : out fieldmode);
    END COMPONENT ;

    COMPONENT Displace
    GENERIC (
```

```
        alpha : complex);
    PORT (
        VacIn : in fieldmode;
        Out1 : out fieldmode);
    END COMPONENT ;

    SIGNAL unnamed_net11 : fieldmode;
    SIGNAL unnamed_net10 : fieldmode;
    SIGNAL unnamed_net9 : fieldmode;
    SIGNAL unnamed_net8 : fieldmode;
    SIGNAL unnamed_net7 : fieldmode;
    SIGNAL unnamed_net6 : fieldmode;
    SIGNAL unnamed_net5 : fieldmode;
    SIGNAL unnamed_net4 : fieldmode;
    SIGNAL unnamed_net3 : fieldmode;
    SIGNAL unnamed_net2 : fieldmode;
    SIGNAL unnamed_net1 : fieldmode;
    SIGNAL w : fieldmode;
BEGIN
-- Architecture statement part
    W_beta : Displace
    GENERIC MAP (
        alpha => beta);
    PORT MAP (
        VacIn => unnamed_net6,
        Out1 => unnamed_net11);

    BS2 : Beamsplitter
    GENERIC MAP (
        theta => theta);
    PORT MAP (
        In1 => unnamed_net11,
        In2 => unnamed_net3,
        Out1 => unnamed_net10,
        Out2 => unnamed_net8);

    BS1 : Beamsplitter
    PORT MAP (
        In1 => unnamed_net4,
        In2 => unnamed_net5,
        Out1 => unnamed_net7,
        Out2 => w);

    P : Phase
    GENERIC MAP (
        phi => phi);
    PORT MAP (
        In1 => unnamed_net10,
        Out1 => unnamed_net9);

    K : KerrCavity
    GENERIC MAP (
        Delta => Delta,
        chi => chi,
        kappa_1 => kappa,
        kappa_2 => kappa);
    PORT MAP (
        In1 => w,
```

```
        Out1 => unnamed_net1,
        In2 => unnamed_net2,
        Out2 => unnamed_net3);

-- Signal assignment part
unnamed_net6 <= VIn1;
unnamed_net2 <= VIn2;
unnamed_net5 <= B;
unnamed_net4 <= A;
NAND_AB <= unnamed_net9;
OUT2 <= unnamed_net8;
UOut2 <= unnamed_net1;
UOut1 <= unnamed_net7;
END netlist;
```

A Pseudo-NAND-Latch

This circuit consists of two subcomponents that each act almost (i.e., for all relevant input conditions) like a NAND logic gate in a symmetric feedback conditions. As is known from electrical circuits this arrangement allows the fabrication of a bi-stable system with memory or state from two systems that have a one-to-one input output behavior. See also *[Mabuchi11]*

```
--pseudo-NAND latch with explicit parameter dependence
entity PseudoNANDLatch is
    generic (Delta, chi, kappa, phi, theta : real;
             beta : complex);

    port (NS, W1, kerr2_extra, NR, W2, kerr1_extra : in fieldmode;
          BS1_1_out, kerr1_out2, OUT2_2, BS1_2_out, kerr2_out2, OUT2_1 : out␣
↪fieldmode);
end PseudoNANDLatch;

architecture latch_netlist of PseudoNANDLatch is
    component PseudoNAND
        generic (Delta, chi, kappa, phi, theta : real;
                 beta : complex);
        port (A, B, W_in, kerr_in2 : in fieldmode;
              uo1, kerr_out1, NAND_AB, OUT2 : out fieldmode);
    end component;

    signal FB12, FB21 : fieldmode;      -- feedback signals

begin
    NAND2 : PseudoNAND
    generic map (
        Delta => Delta, chi => chi, kappa => kappa, phi => phi, theta => theta, beta␣
↪=> beta);
    port map (
        A => NR, B => FB12, W_in => W2, kerr_in2 => kerr2_extra,
        uo1 => BS1_2_out, kerr_out1 => kerr2_out2, NAND_AB => FB21, OUT2 => OUT2_2);

    NAND1 : PseudoNAND
    generic map (
        Delta => Delta, chi => chi, kappa => kappa, phi => phi, theta => theta, beta␣
↪=> beta);
    port map (
```

```
        A => NS, B => FB21, W_in => W1, kerr_in2 => kerr1_extra,
        uo1 => BS1_1_out, kerr_out1 => kerr1_out2, NAND_AB => FB12, OUT2 => OUT2_1);
end latch_netlist;
```

CHAPTER 7

Parsing QHDL

Given a QHDL-file `my_circuit.qhdl` which contains an entity named `MyEntity` (Note again the CamelCaseConvention for entity names!), we have two options for the final python circuit model file:

1. We can compile it to an output in the local directory. To do this run in the shell:

   ```
   $QNET/bin/parse_qhdl.py -f my_circuit.qhdl -l
   ```

2. We can compile it and install it within the module *qnet.circuit_components*. To do this run in the shell:

   ```
   $QNET/bin/parse_qhdl.py -f my_circuit.qhdl -L
   ```

In either case the output file will be named based on a CamelCase to lower_case_with_underscore convention with a `_cc` suffix to the name. I.e., for the above example `MyEntity` will become `my_entity_cc.py`. In the case of entity names with multiple subsequent capital letters such as `PseudoNAND` the convention is to only add underlines before the first and the last of the capitalized group, i.e. the output would be written to `pseudo_nand_cc.py`.

CHAPTER 8

Symbolic Analysis and Simulation

Symbolic Analysis of the Pseudo NAND gate and the Pseudo NAND SR-Latch

Pseudo NAND gate

In[1]:

```
from qnet.algebra.circuit_algebra import *
```

In[2]:

```
from qnet.circuit_components import pseudo_nand_cc as nand

# real parameters
kappa = symbols('kappa', positive = True)
Delta, chi, phi, theta = symbols('Delta, chi, phi, theta', real = True)

# complex parameters
A, B, beta = symbols('A, B, beta')

N = nand.PseudoNAND('N', kappa=kappa, Delta=Delta, chi=chi, phi=phi, theta=theta,␣
↪beta=beta)
N
```

Out[2]:

N

Circuit Analysis of Pseudo NAND gate

In[3]:

```
N.creduce()
```

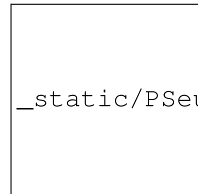
Out[3]:

$$\left(\mathbf{1}_1 \boxplus ((\mathbf{1}_1 \boxplus ((\text{N.P} \boxplus \mathbf{1}_1) \lhd \text{N.BS2} \lhd (W(\beta) \boxplus \mathbf{1}_1))) \lhd \mathbf{P}_\sigma \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 1 \end{pmatrix} \lhd (\text{N.K} \boxplus \mathbf{1}_1)\right) \lhd (\text{N.BS1} \boxplus \mathbf{1}_2) \lhd \mathbf{P}_\sigma \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 3 & 2 \end{pmatrix}$$

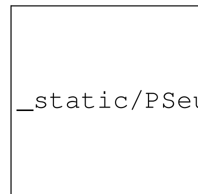
In[4]:

```
# yields a single block
N.show()

# decompose into sub components
N.creduce().show()
```



_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_00.png



_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_01.png

SLH model

In[5]:

```
NSLH = N.coherent_input(A, B, 0, 0).toSLH()
NSLH
```

Out[5]:

$$\left(\begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 & 0 \\ 0 & 0 & e^{\imath\phi}\cos(\theta) & -e^{\imath\phi}\sin(\theta) \\ 0 & 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}, \begin{pmatrix} \frac{1}{2}\sqrt{2}A - \frac{1}{2}\sqrt{2}B \\ \left(\frac{1}{2}\sqrt{2}A + \frac{1}{2}\sqrt{2}B\right) + \sqrt{\kappa}a_{\text{N.K}} \\ \beta e^{\imath\phi}\cos(\theta) - \sqrt{\kappa}e^{\imath\phi}\sin(\theta)\, a_{\text{N.K}} \\ \beta\sin(\theta) + \sqrt{\kappa}\cos(\theta)\, a_{\text{N.K}} \end{pmatrix}, \frac{1}{2}\imath\left(-\left(\frac{1}{2}\sqrt{2}A\sqrt{\kappa} + \frac{1}{2}\sqrt{2}B\sqrt{\kappa}\right)a_{\text{N.K}}^\dagger + \right.$$

Heisenberg equation of motion of the mode operator a

In[6]:

```
s = N.space
a = Destroy(s)
a
```

Out[6]:

$$a_{\text{N.K}}$$

In[7]:

```
NSLH.symbolic_heisenberg_eom(a).expand().simplify_scalar()
```

Out[7]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-A-B\right)-\left(\imath\Delta+\kappa\right)a_{\rm N.K}-2\imath\chi a^{\dagger}_{\rm N.K}a_{\rm N.K}a_{\rm N.K}$$

Super operator algebra: The system's liouvillian and a re-derivation of the eom for a via the super-operator adjoint of the liouvillian.

In[8]:

```
LLN = NSLH.symbolic_liouvillian().expand().simplify_scalar()
LLN
```

Out[8]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(A+B\right)\text{spost}\left[a^{\dagger}_{\rm N.K}\right]+\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-\overline{A}-\overline{B}\right)\text{spost}\left[a_{\rm N.K}\right]+\imath\chi\text{spost}\left[a^{\dagger}_{\rm N.K}a^{\dagger}_{\rm N.K}a_{\rm N.K}a_{\rm N.K}\right]+\left(\imath\Delta-\kappa\right)\text{spost}\left[a^{\dagger}_{\rm N.K}a_{\rm N.K}\right]+$$

In[9]:

```
(LLN.superadjoint() * a).expand().simplify_scalar()
```

Out[9]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-A-B\right)-\left(\imath\Delta+\kappa\right)a_{\rm N.K}-2\imath\chi a^{\dagger}_{\rm N.K}a_{\rm N.K}a_{\rm N.K}$$

A full Pseudo-NAND SR-Latch

In[10]:

```
N1 = nand.PseudoNAND('N_1', kappa=kappa, Delta=Delta, chi=chi, phi=phi, theta=theta,␣
↪beta=beta)
N2 = nand.PseudoNAND('N_2', kappa=kappa, Delta=Delta, chi=chi, phi=phi, theta=theta,␣
↪beta=beta)

# NAND gates in mutual feedback configuration
NL = (N1 + N2).feedback(2, 4).feedback(5, 0).coherent_input(A, 0, 0, B, 0, 0)
NL
```

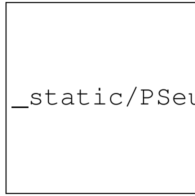
Out[10]:

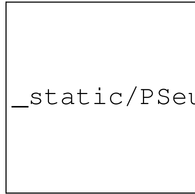
$$\left\lfloor\left(\mathbf{1}_3\boxplus{\rm N}_2\right)\lhd\left(\left(\mathbf{P}_\sigma\begin{pmatrix}0&1&2&3\\0&1&3&2\end{pmatrix}\lhd{\rm N}_1\right)\boxplus\mathbf{1}_3\right)\right\rfloor_{5\to0}\lhd\left(W(A)\boxplus\mathbf{1}_2\boxplus W(B)\boxplus\mathbf{1}_2\right)$$

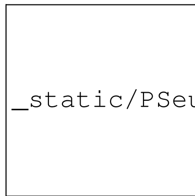
The circuit algebra simplification rules have already eliminated one of the two feedback operations in favor or a series product.

In[11]:

```
NL.show()
NL.creduce().show()
NL.creduce().creduce().show()
```


_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_02.png


_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_03.png


_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_04.png

SLH model

In[12]:

```
NLSLH = NL.toSLH().expand().simplify_scalar()
NLSLH
```

Out[12]:

$$\left(\begin{pmatrix} -\frac{1}{2}\sqrt{2} & 0 & 0 & 0 & \frac{1}{2}\sqrt{2}e^{\imath\phi}\cos\left(\theta\right) & -\frac{1}{2}\sqrt{2}e^{\imath\phi}\sin\left(\theta\right) \\ \frac{1}{2}\sqrt{2} & 0 & 0 & 0 & \frac{1}{2}\sqrt{2}e^{\imath\phi}\cos\left(\theta\right) & -\frac{1}{2}\sqrt{2}e^{\imath\phi}\sin\left(\theta\right) \\ 0 & \sin\left(\theta\right) & \cos\left(\theta\right) & 0 & 0 & 0 \\ 0 & \frac{1}{2}\sqrt{2}e^{\imath\phi}\cos\left(\theta\right) & -\frac{1}{2}\sqrt{2}e^{\imath\phi}\sin\left(\theta\right) & -\frac{1}{2}\sqrt{2} & 0 & 0 \\ 0 & \frac{1}{2}\sqrt{2}e^{\imath\phi}\cos\left(\theta\right) & -\frac{1}{2}\sqrt{2}e^{\imath\phi}\sin\left(\theta\right) & \frac{1}{2}\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sin\left(\theta\right) & \cos\left(\theta\right) \end{pmatrix}, \begin{pmatrix} \frac{1}{2}\sqrt{2}\left(-A+\beta e^{\imath\phi}\cos\left(\theta\right)\right) - \\ \frac{1}{2}\sqrt{2}\left(A+\beta e^{\imath\phi}\cos\left(\theta\right)\right) + \sqrt{\kappa}a_{N} \\ \beta\sin\left(\theta\right) + \sqrt{\kappa} \\ \frac{1}{2}\sqrt{2}\left(-B+\beta e^{\imath\phi}\cos\left(\theta\right)\right) - \\ \frac{1}{2}\sqrt{2}\left(B+\beta e^{\imath\phi}\cos\left(\theta\right)\right) - \frac{1}{2}\sqrt{2} \\ \beta\sin\left(\theta\right) + \sqrt{\kappa} \end{pmatrix}\right.$$

Heisenberg equations of motion for the mode operators

In[13]:

```
NL.space
```

Out[13]:

$$\mathrm{N}_1.\mathrm{K} \otimes \mathrm{N}_2.\mathrm{K}$$

In[14]:

```
s1, s2 = NL.space.operands
a1 = Destroy(s1)
a2 = Destroy(s2)
```

In[15]:

```
da1dt = NLSLH.symbolic_heisenberg_eom(a1).expand().simplify_scalar()
da1dt
```

Out[15]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-A-\beta e^{\imath\phi}\cos\left(\theta\right)\right)-\left(\imath\Delta+\kappa\right)a_{\mathrm{N}_1.\mathrm{K}}+\frac{1}{2}\sqrt{2}\kappa e^{\imath\phi}\sin\left(\theta\right)a_{\mathrm{N}_2.\mathrm{K}}-2\imath\chi a^{\dagger}_{\mathrm{N}_1.\mathrm{K}}a_{\mathrm{N}_1.\mathrm{K}}a_{\mathrm{N}_1.\mathrm{K}}$$

In[16]:

```
da2dt = NLSLH.symbolic_heisenberg_eom(a2).expand().simplify_scalar()
da2dt
```

Out[16]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-B-\beta e^{\imath\phi}\cos\left(\theta\right)\right)+\frac{1}{2}\sqrt{2}\kappa e^{\imath\phi}\sin\left(\theta\right)a_{\mathrm{N}_1.\mathrm{K}}-\left(\imath\Delta+\kappa\right)a_{\mathrm{N}_2.\mathrm{K}}-2\imath\chi a^{\dagger}_{\mathrm{N}_2.\mathrm{K}}a_{\mathrm{N}_2.\mathrm{K}}a_{\mathrm{N}_2.\mathrm{K}}$$

Show Exchange-Symmetry of the Pseudo NAND latch Liouvillian super operator

Simultaneously exchanging the degrees of freedom and the coherent input amplitudes leaves the liouvillian unchanged.

In[17]:

```
C = symbols('C')
LLNL = NLSLH.symbolic_liouvillian().expand().simplify_scalar()
LLNL
```

Out[17]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(A+\beta e^{\imath\phi}\cos\left(\theta\right)\right)\mathrm{spost}\left[a^{\dagger}_{\mathrm{N}_1.\mathrm{K}}\right]+\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(B+\beta e^{\imath\phi}\cos\left(\theta\right)\right)\mathrm{spost}\left[a^{\dagger}_{\mathrm{N}_2.\mathrm{K}}\right]+\frac{\sqrt{2}\sqrt{\kappa}\left(-e^{\imath\phi}\overline{A}-\cos\left(\theta\right)\overline{\beta}\right)}{2e^{\imath\phi}}\mathrm{spost}\left[a_{\mathrm{N}_1.\mathrm{K}}\right]$$

In[18]:

```
C = symbols('C')
(LLNL.substitute({A:C}).substitute({B:A}).substitute({C:B}) - LLNL.substitute({s1:s2,
↪s2:s1}).expand().simplify_scalar()).expand().simplify_scalar()
```

Out[18]:

$$\hat{0}$$

Numerical Analysis via QuTiP

Input-Output Logic of the Pseudo-NAND Gate

In[19]:

```
NSLH.space
```

Out[19]:

$$\mathrm{N.K}$$

In[20]:

```
NSLH.space.dimension = 75
```

Numerical parameters taken from

Mabuchi, H. (2011). Nonlinear interferometry approach to photonic sequential logic. Appl. Phys. Lett. 99, 153103 (2011)

In[21]:

```
# numerical values for simulation

alpha = 22.6274                              # logical 'one' amplitude

numerical_vals = {
                  beta: -34.289-11.909j,      # bias input for pseudo-nands
                  kappa: 25.,                 # Kerr-Cavity mirror couplings
                  Delta: 50.,                 # Kerr-Cavity Detuning
                  chi : -50./60.,             # Kerr-Non-Linear coupling coefficient
                  theta: 0.891,               # pseudo-nand beamsplitter mixing angle
                  phi: 2.546,                 # pseudo-nand corrective phase
    }
```

In[22]:

```
NSLHN = NSLH.substitute(numerical_vals)
NSLHN
```

Out[22]:

$$\left(\begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 & 0 \\ 0 & 0 & 0.628634640249695e^{2.546\imath} & -0.777700770912654e^{2.546\imath} \\ 0 & 0 & 0.777700770912654 & 0.628634640249695 \end{pmatrix}, \begin{pmatrix} \frac{1}{2}\sqrt{2}A - \frac{1}{2}\sqrt{2}B \\ \left(\frac{1}{2}\sqrt{2}A + \frac{1}{2}\sqrt{2}B\right) + 5 \\ 0.628634640249695\left(-34.289 - 11.909\imath\right)e^{2.546\imath} \\ -\left(26.666581733824 + 9.2616384807988\imath\right. \end{pmatrix}\right.$$

In[23]:

```
input_configs = [
            (0,0),
            (1, 0),
            (0, 1),
            (1, 1)
          ]
```

In[24]:

```
Lout = NSLHN.L[2,0]
Loutqt = Lout.to_qutip()
times = arange(0, 1., 0.01)
psi0 = qutip.basis(N.space.dimension, 0)
datasets = {}
for ic in input_configs:
    H, Ls = NSLHN.substitute({A: ic[0]*alpha, B: ic[1]*alpha}).HL_to_qutip()
    data = qutip.mcsolve(H, psi0, times, Ls, [Loutqt], ntraj = 1)
    datasets[ic] = data.expect[0]
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

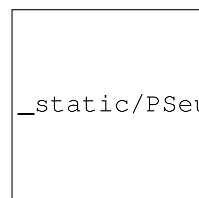
```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

In[25]:

```
figure(figsize=(10, 8))
for ic in input_configs:
    plot(times, real(datasets[ic])/alpha, '-', label = str(ic) + ", real")
    plot(times, imag(datasets[ic])/alpha, '--', label = str(ic) + ", imag")
legend()
xlabel('Time $t$', size = 20)
ylabel(r'$\langle L_out \rangle$ in logic level units', size = 20)
title('Pseudo NAND logic, stochastically simulated time \n dependent output␣
↪amplitudes for different inputs.', size = 20)
```

Out[25]:

```
<matplotlib.text.Text at 0x1100b7dd0>
```



_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_05.png

Pseudo NAND latch memory effect

In[26]:

```
NLSLH.space
```

Out[26]:

$$\mathrm{N}_1.\mathrm{K} \otimes \mathrm{N}_2.\mathrm{K}$$

In[27]:

```
s1, s2 = NLSLH.space.operands
s1.dimension = 75
s2.dimension = 75
NLSLH.space.dimension
```

Out[27]:

```
5625
```

In[28]:

```
NLSLHN = NLSLH.substitute(numerical_vals)
NLSLHN
```

Out[28]:

$$\left(\left(\begin{matrix} -\frac{1}{2}\sqrt{2} & 0 & 0 & 0 & 0.314317320124847\sqrt{2}e^{2.546\imath} & -0.388850 \\ \frac{1}{2}\sqrt{2} & 0 & 0 & 0 & 0.314317320124847\sqrt{2}e^{2.546\imath} & -0.388850 \\ 0 & 0.777700770912654 & 0.628634640249695 & 0 & 0 & \\ 0 & 0.314317320124847\sqrt{2}e^{2.546\imath} & -0.388850385456327\sqrt{2}e^{2.546\imath} & -\frac{1}{2}\sqrt{2} & 0 & \\ 0 & 0.314317320124847\sqrt{2}e^{2.546\imath} & -0.388850385456327\sqrt{2}e^{2.546\imath} & \frac{1}{2}\sqrt{2} & 0 & \\ 0 & 0 & 0 & 0 & 0.777700770912654 & 0.628 \end{matrix}\right.\right.$$

In[29]:

```
input_configs = {
            "SET": (1, 0),
            "RESET": (0, 1),
            "HOLD": (1, 1)
          }

models = {k: NLSLHN.substitute({A:v[0]*alpha, B:v[1]*alpha}).HL_to_qutip() for k, v in input_configs.items()}
```

In[30]:

```
a1, a2 = Destroy(s1), Destroy(s2)
observables = [a1.dag()*a1, a2.dag()*a2]
observables_qt = [o.to_qutip(full_space = NLSLH.space) for o in observables]
```

In[31]:

```
def model_sequence_single_trajectory(models, durations, initial_state, dt):
    """
    Solve a sequence of constant QuTiP open system models (H_i, [L_1_i, L_2_i, ...])
    via Quantum Monte-Carlo. Each model is valid for a duration deltaT_i and the initial state for
    is given by the previous model's final state.
    The function returns an array with the times and an array with the states at each time.

    :param models: Sequence of models given as tuples: (H_j, [L1j,L2j,...])
    :type models: Sequence of tuples
    :param durations: Sequence of times
    :type durations: Sequence of float
    :param initial_state: Overall initial state
    :type initial_state: qutip.Qobj
    :param dt: Sampling interval
    :type dt: float
    :return: times, states
    :rtype: tuple((numpy.ndarray, numpy.ndarray)
    """
    totalT = 0
    totalTimes = array([])
    totalStates = array([])
    current_state = initial_state

    for j, (model, deltaT) in enumerate(zip(models, durations)):
        print "Solving step {}/{} of model sequence".format(j + 1, len(models))
        HQobj, LQObjs = model
        times = arange(0, deltaT, dt)
        data = qutip.mcsolve(HQobj, current_state, times, LQObjs, [], ntraj = 1, options = qutip.Odeoptions(gui = False))
```

```
        # concatenate states
        totalStates = np.hstack((totalStates,data.states.flatten()))
        current_state = data.states.flatten()[-1]
        # concatenate times
        totalTimes = np.hstack((totalTimes, times + totalT))
        totalT += times[-1]

    return totalTimes, totalStates
```

In[32]:

```
durations = [.5, 1., .5, 1.]
model_sequence = [models[v] for v in ['SET', 'HOLD', 'RESET', 'HOLD']]
initial_state = qutip.tensor(qutip.basis(s1.dimension, 0), qutip.basis(s2.dimension,␣
↪0))
```

In[33]:

```
times, data = model_sequence_single_trajectory(model_sequence, durations, initial_
↪state, 5e-3)
```

```
Solving step 1/4 of model sequence
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
Solving step 2/4 of model sequence
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
Solving step 3/4 of model sequence
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
Solving step 4/4 of model sequence
```

```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

In[34]:

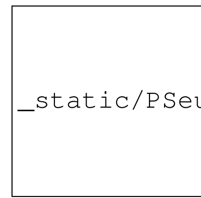
```
datan1 = qutip.expect(observables_qt[0], data)
datan2 = qutip.expect(observables_qt[1], data)
```

In[36]:

```
figsize(10,6)
plot(times, datan1)
plot(times, datan2)
for t in cumsum(durations):
    axvline(t, color = "r")
xlabel("Time $t$", size = 20)
ylabel("Intra-cavity Photon Numbers", size = 20)
legend((r"$\langle n_1 \rangle :math:`", r"`\langle n_2 \rangle $"), loc = 'lower␣
↪right')
title("SET - HOLD - RESET - HOLD sequence for $\overline{SR}$-latch", size = 20)
```

Out[36]:

```
<matplotlib.text.Text at 0x1121d8990>
```



_static/PSeudoNANDAnalysis_files/PseudoNANDAnalysis_fig_06.png

CHAPTER 9

References

CHAPTER 10

API

qnet package

The *qnet* package exposes all of QNET's functionality for easy interactive or programmative use.

Specifically, the subpackages for the following parts of QNET are directly available:

- Symbolic quantum and circuit algebra as *qnet.algebra*
- Printers for symbolic symbolic expressions as *qnet.printing*
- A library of circuit components as *qnet.cc* (as a shorthand for the *circuit_components* subpackage)
- Conversion utilities to Sympy and Numpy as *qnet.convert*
- Parsing utilities for the QHDL language, as *qnet.qhdl*
- Miscellaneous additional tools, as *qnet.misc*

For interactive usage, the package should be initialized as follows:

```
>>> import qnet
>>> qnet.init_printing()
```

Note that most subpackages in turn expose their functionality through a "flat" API. That is, instead of

```
from qnet.algebra.operator_algebra import LocalOperator
from qnet.circuit_components.displace_cc import Displace
```

the two objects may be more succintly imported from a higher level namespace as

```
import qnet  # required for qnet.cc to work
from qnet.algebra import LocalOperator
from qnet.cc import Displace
```

In an interactive context (and only there!), a star import such as

```
from qnet.algebra import *
```

may be useful.

The flat API is defined via the __all__ attribute of each subpackage (see each package's documentation).

Internally, the flat API (or star imports) must never be used.

Subpackages:

qnet.algebra package

Submodules:

qnet.algebra.abstract_algebra module

The abstract algebra package provides a basic interface for defining custom Algebras.

See *The Abstract Algebra module* for design details and usage.

Summary

Exceptions:

AlgebraError	Base class for all errors concerning the mathematical definitions and rules of an algebra.
AlgebraException	Base class for all errors concerning the mathematical definitions and rules of an algebra.
CannotSimplify	Raised when an expression cannot be further simplified
WrongSignatureError	Raised when an operation is instantiated with operands of the wrong signature.

Classes:

Expression	Abstract class for QNET Expressions.
Operation	Base class for all "operations", i.e.

Functions:

all_symbols	Return all all_symbols featured within an expression.
assoc	Associatively expand out nested arguments of the flat class.
cache_attr	A method decorator that caches the result of a method in an attribute, intended for e.g.
check_idempotent_create	Check that an expression is 'idempotent'
extra_binary_rules	Context manager that temporarily adds the given rules to *cls* (to be processed by *match_replace_binary*.
extra_rules	Context manager that temporarily adds the given rules to *cls* (to be processed by *match_replace*.
filter_neutral	Remove occurrences of a neutral element from the argument/operand list, if that list has at least two elements.

Continued on next page

Table 10.3 – continued from previous page

idem	Remove duplicate arguments and order them via the cls's order_key key object/function.
match_replace	Match and replace a full operand specification to a function that provides a replacement for the whole expression or raises a *CannotSimplify* exception.
match_replace_binary	Similar to func:*match_replace*, but for arbitrary length operations, such that each two pairs of subsequent operands are matched pairwise.
no_instance_caching	Temporarily disable the caching of instances through
no_rules	Context manager that temporarily disables all rules (processed by *match_replace* or *match_replace_binary*) for the given *cls*.
orderby	Re-order arguments via the class's `order_key` key object/function.
set_union	Similar to `sum()`, but for sets.
simplify	Recursively re-instantiate the expression, while applying all of the
substitute	Substitute symbols or (sub-)expressions with the given replacements and
temporary_instance_cache	Use a temporary cache for instances obtained from the *create* method of the given *cls*.

`__all__`: *AlgebraError*, *AlgebraException*, *CannotSimplify*, *Expression*, *Operation*, SCALAR_TYPES, *WrongSignatureError*, *all_symbols*, *extra_binary_rules*, *extra_rules*, *no_instance_caching*, *no_rules*, *set_union*, *simplify*, *substitute*, *temporary_instance_cache*

Reference

exception `qnet.algebra.abstract_algebra.`**`AlgebraException`**

Bases: `Exception`

Base class for all errors concerning the mathematical definitions and rules of an algebra.

exception `qnet.algebra.abstract_algebra.`**`AlgebraError`**

Bases: *qnet.algebra.abstract_algebra.AlgebraException*

Base class for all errors concerning the mathematical definitions and rules of an algebra.

exception `qnet.algebra.abstract_algebra.`**`CannotSimplify`**

Bases: *qnet.algebra.abstract_algebra.AlgebraException*

Raised when an expression cannot be further simplified

exception `qnet.algebra.abstract_algebra.`**`WrongSignatureError`**

Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Raised when an operation is instantiated with operands of the wrong signature.

`qnet.algebra.abstract_algebra.`**`cache_attr`**(*attr*)

A method decorator that caches the result of a method in an attribute, intended for e.g. __str__

```
>>> class MyClass():
...     def __init__(self):
...         self._str = None
```

```
...
...     @cache_attr('_str')
...     def __str__(self):
...         return "MyClass"
>>> a = MyClass()
>>> a._str  # None
>>> str(a)
'MyClass'
>>> a._str
'MyClass'
```

class qnet.algebra.abstract_algebra.**Expression**(**args*, ***kwargs*)
Bases: object

Abstract class for QNET Expressions. All algebraic objects are either scalars (numbers or Sympy expressions) or instances of Expression.

Expressions should generally be instantiated using the *create* class method, which takes into account the algebraic properties of the Expression and and applies simplifications. It also uses memoization to cache all known (sub-)expression. This is possible because expressions are intended to be immutable. Any changes to an expression should be made through e.g. substitute(), which returns a new modified expression.

Every expression has a well-defined list of positional and keyword arguments that uniquely determine the expression and that may be accessed through the *args* and *kwargs* property. That is,

```
expr.__class__(*expr.args, **expr.kwargs)
```

will return and object identical to *expr*.

Class Attributes **instance_caching** (*bool*) – Flag to indicate whether the *create* class method should cache the instantiation of instances

instance_caching = True

classmethod create(**args*, ***kwargs*)
Instead of directly instantiating, it is recommended to use create, which applies simplifications to the args and keyword arguments according to the *_simplifications* class attribute, and returns an appropriate object (which may or may not be an instance of the original class)

args
The tuple of positional arguments for the instantiation of the Expression

kwargs
The dictionary of keyword-only arguments for the instantiation of the Expression

minimal_kwargs
A "minimal" dictionary of keyword-only arguments, i.e. a subsect of *kwargs* that may exclude default options

substitute(*var_map*)
Substitute all_symbols for other expressions.

Parameters **var_map** (*dict*) – Dictionary with entries of the form {expr: substitution}

simplify(*rules=None*)
Recursively re-instantiate the expression, while applying all of the given *rules* to all encountered (sub-) expressions

all_symbols()
Set of all_symbols contained within the expression.

__ne__(*other*)
If it is well-defined (i.e. boolean), simply return the negation of `self.__eq__(other)` Otherwise return NotImplemented.

__getstate__()
state to be pickled

qnet.algebra.abstract_algebra.**check_idempotent_create**(*expr*)
Check that an expression is 'idempotent'

qnet.algebra.abstract_algebra.**substitute**(*expr*, *var_map*)
Substitute symbols or (sub-)expressions with the given replacements and re-evalute the result

Parameters

- **expr** – The expression in which to perform the substitution
- **var_map** (*dict*) – The substitution dictionary.

qnet.algebra.abstract_algebra.**simplify**(*expr*, *rules=None*)
Recursively re-instantiate the expression, while applying all of the given *rules* to all encountered (sub-) expressions

Parameters

- **expr** – Any Expression or scalar object
- **rules** (*list*) – A list of tuples `(pattern, replacement)` where *rule* is an instance of `Pattern`) and *replacement* is a callable. The pattern will be matched against any expression that is encountered during the re-instantiation. If the *pattern* matches, then the (sub-)expression is replaced by the result of calling *replacement* while passing any wildcards from *pattern* as keyword arguments. If *replacement* raises *CannotSimplify*, it will be ignored

Note: Instead of or in addition to passing *rules*, *simplify* can often be combined with e.g. *extra_rules* / *extra_binary_rules* context managers. If a simplification can be handled through these context managers, this is usually more efficient than an equivalent rule. However, both really are complemetary: the rules defined in the context managers are applied *before* instantation (hence these these patterns are instantiated through *pattern_head*). In contrast, the patterns defined in *rules* are applied against instantiated expressions.

qnet.algebra.abstract_algebra.**set_union**(**sets*)
Similar to `sum()`, but for sets. Generate the union of an arbitrary number of set arguments.

qnet.algebra.abstract_algebra.**all_symbols**(*expr*)
Return all all_symbols featured within an expression.

class qnet.algebra.abstract_algebra.**Operation**(**operands*, ***kwargs*)
Bases: *qnet.algebra.abstract_algebra.Expression*

Base class for all "operations", i.e. Expressions that act algebraically on other expressions (their "operands").

Operations differ from more general Expressions by the convention that the arguments of the Operator are exactly the operands (which must be members of the algebra!) Any other parameters (non-operands) that may be required must be given as keyword-arguments.

operands
Tuple of operands of the operation

args
Alias for operands

qnet.algebra.abstract_algebra.**assoc**(*cls*, *ops*, *kwargs*)
Associatively expand out nested arguments of the flat class. E.g.:

```
>>> class Plus(Operation):
...     _simplifications = [assoc, ]
>>> Plus.create(1,Plus(2,3))
Plus(1, 2, 3)
```

qnet.algebra.abstract_algebra.**idem**(*cls*, *ops*, *kwargs*)
Remove duplicate arguments and order them via the cls's order_key key object/function. E.g.:

```
>>> class Set(Operation):
...     order_key = lambda val: val
...     _simplifications = [idem, ]
>>> Set.create(1,2,3,1,3)
Set(1, 2, 3)
```

qnet.algebra.abstract_algebra.**orderby**(*cls*, *ops*, *kwargs*)
Re-order arguments via the class's `order_key` key object/function. Use this for commutative operations: E.g.:

```
>>> class Times(Operation):
...     order_key = lambda val: val
...     _simplifications = [orderby, ]
>>> Times.create(2,1)
Times(1, 2)
```

qnet.algebra.abstract_algebra.**filter_neutral**(*cls*, *ops*, *kwargs*)
Remove occurrences of a neutral element from the argument/operand list, if that list has at least two elements. To use this, one must also specify a neutral element, which can be anything that allows for an equality check with each argument. E.g.:

```
>>> class X(Operation):
...     neutral_element = 1
...     _simplifications = [filter_neutral, ]
>>> X.create(2,1,3,1)
X(2, 3)
```

qnet.algebra.abstract_algebra.**match_replace**(*cls*, *ops*, *kwargs*)
Match and replace a full operand specification to a function that provides a replacement for the whole expression or raises a *CannotSimplify* exception. E.g.

First define an operation:

```
>>> class Invert(Operation):
...     _rules = []
...     _simplifications = [match_replace, ]
```

Then some _rules:

```
>>> A = wc("A")
>>> A_float = wc("A", head=float)
>>> Invert_A = pattern(Invert, A)
>>> Invert._rules += [
...     (pattern_head(Invert_A), lambda A: A),
...     (pattern_head(A_float), lambda A: 1./A),
... ]
```

Check rule application:

```
>>> print(srepr(Invert.create("hallo")))  # matches no rule
Invert('hallo')
>>> Invert.create(Invert("hallo"))        # matches first rule
'hallo'
>>> Invert.create(.2)                     # matches second rule
5.0
```

A pattern can also have the same wildcard appear twice:

```
>>> class X(Operation):
...     _rules = [
...         (pattern_head(A, A), lambda A: A),
...     ]
...     _simplifications = [match_replace, ]
>>> X.create(1,2)
X(1, 2)
>>> X.create(1,1)
1
```

qnet.algebra.abstract_algebra.**match_replace_binary**(*cls*, *ops*, *kwargs*)

Similar to func:*match_replace*, but for arbitrary length operations, such that each two pairs of subsequent operands are matched pairwise.

```
>>> A = wc("A")
>>> class FilterDupes(Operation):
...     _binary_rules = [(pattern_head(A,A), lambda A: A), ]
...     _simplifications = [match_replace_binary, assoc]
...     neutral_element = 0
>>> FilterDupes.create(1,2,3,4)        # No duplicates
FilterDupes(1, 2, 3, 4)
>>> FilterDupes.create(1,2,2,3,4)      # Some duplicates
FilterDupes(1, 2, 3, 4)
```

Note that this only works for *subsequent* duplicate entries:

```
>>> FilterDupes.create(1,2,3,2,4)      # No *subsequent* duplicates
FilterDupes(1, 2, 3, 2, 4)
```

Any operation that uses binary reduction must be associative and define a neutral element. The binary rules must be compatible with associativity, i.e. there is no specific order in which the rules are applied to pairs of operands.

qnet.algebra.abstract_algebra.**no_instance_caching**()

Temporarily disable the caching of instances through *Expression.create*

qnet.algebra.abstract_algebra.**temporary_instance_cache**(*cls*)

Use a temporary cache for instances obtained from the *create* method of the given *cls*. That is, no cached instances from outside of the managed context will be used within the managed context, and vice versa

qnet.algebra.abstract_algebra.**extra_rules**(*cls*, *rules*)

Context manager that temporarily adds the given rules to *cls* (to be processed by *match_replace*. Implies *temporary_instance_cache*.

qnet.algebra.abstract_algebra.**extra_binary_rules**(*cls*, *rules*)

Context manager that temporarily adds the given rules to *cls* (to be processed by *match_replace_binary*. Implies *temporary_instance_cache*.

qnet.algebra.abstract_algebra.**no_rules**(*cls*)

Context manager that temporarily disables all rules (processed by *match_replace* or *match_replace_binary*) for

the given *cls*. Implies *temporary_instance_cache*.

qnet.algebra.circuit_algebra module

This module defines the circuit algebra for quantum optical feedback and feedforward circuits in the zero-internal time-delay limit. For more details see *The Circuit Algebra module*.

References:

Summary

Exceptions:

`CannotConvertToABCD`	Is raised when a circuit algebra object cannot be converted to a concrete ABCD object.
`CannotConvertToSLH`	Is raised when a circuit algebra object cannot be converted to a concrete SLH object.
`CannotEliminateAutomatically`	Raised when attempted automatic adiabatic elimination fails.
`CannotVisualize`	Is raised when a circuit algebra object cannot be visually represented.
`IncompatibleBlockStructures`	Is raised when a circuit decomposition into a block-structure is requested that is icompatible with the actual block structure of the circuit expression.
`WrongCDimError`	Is raised when two object are tried to joined together in series but have different channel dimensions.

Classes:

`ABCD`	ABCD model class in amplitude representation.
`CPermutation`	The channel permuting circuit.
`Circuit`	Abstract base class for the circuit algebra elements.
`CircuitSymbol`	Circuit Symbol object, parametrized by an identifier (name) and channel dimension.
`Concatenation`	The concatenation product circuit operation.
`Feedback`	The circuit feedback operation applied to a circuit of channel dimension > 1 and an from an output port index to an input port index.
`SLH`	SLH class to encapsulate an open system model that is parametrized as
`SeriesInverse`	Symbolic series product inversion operation.
`SeriesProduct`	The series product circuit operation.

Functions:

`FB`	Wrapper for :py:class:Feedback: but with additional default values.
`P_sigma`	Create a channel permutation circuit for the given index image values.

Continued on next page

Table 10.6 – continued from previous page

check_cdims	Check that all operands (*ops*) have equal channel dimension.
cid	Return the circuit identity for n channels.
circuit_identity	Return the circuit identity for n channels.
connect	Connect a list of components according to a list of connections.
eval_adiabatic_limit	Compute the limiting SLH model for the adiabatic approximation.
extract_signal	Create a permutation that maps the k-th (zero-based) element to the last element, while preserving the relative order of all other elements.
extract_signal_circuit	Create a channel permutation circuit that maps the k-th (zero-based) input to the last output, while preserving the relative order of all other channels.
getABCD	Return the A, B, C, D and (a, c) matrices that linearize an SLH model about a coherent displacement amplitude a0.
get_common_block_structure	For two block structures `aa = (a1, a2, ..., an)`, `bb = (b1, b2, ..., bm)` generate the maximal common block structure so that every block from aa and bb is contained in exactly one block of the resulting structure.
map_signals	For a given {input:output} mapping in form of a dictionary, generate the permutation that achieves the specified mapping while leaving the relative order of all non-specified elements intact.
map_signals_circuit	For a given {input:output} mapping in form of a dictionary, generate the channel permutating circuit that achieves the specified mapping while leaving the relative order of all non-specified channels intact.
move_drive_to_H	For the given *slh* model, move inhomogeneities in the Lindblad operators (resulting from the presence of a coherent drive, see *Displace*) to the Hamiltonian.
pad_with_identity	Pad a circuit by 'inserting' an n-channel identity circuit at index k.
prepare_adiabatic_limit	Prepare the adiabatic elimination procedure for an SLH object with
try_adiabatic_elimination	Attempt to automatically carry out the adiabatic elimination procedure on slh with scaling parameter k.

`__all__`: *ABCD*, *CIdentity*, *CPermutation*, *CannotConvertToABCD*, *CannotConvertToSLH*, *CannotEliminateAutomatically*, *CannotVisualize*, *Circuit*, *CircuitSymbol*, *CircuitZero*, *Concatenation*, *FB*, *Feedback*, *IncompatibleBlockStructures*, *P_sigma*, *SLH*, *SeriesInverse*, *SeriesProduct*, *WrongCDimError*, *cid*, *cid_1*, *circuit_identity*, *connect*, *eval_adiabatic_limit*, *extract_signal*, *extract_signal_circuit*, *getABCD*, *get_common_block_structure*, *map_signals*, *map_signals_circuit*, *move_drive_to_H*, *pad_with_identity*, *prepare_adiabatic_limit*, *try_adiabatic_elimination*

Module data:

`qnet.algebra.circuit_algebra.`**`CIdentity`**

`qnet.algebra.circuit_algebra.`**`CircuitZero`**

`qnet.algebra.circuit_algebra.`**`cid_1`**

Reference

exception qnet.algebra.circuit_algebra.**CannotConvertToSLH**
Bases: *qnet.algebra.abstract_algebra.AlgebraException*

Is raised when a circuit algebra object cannot be converted to a concrete SLH object.

exception qnet.algebra.circuit_algebra.**CannotConvertToABCD**
Bases: *qnet.algebra.abstract_algebra.AlgebraException*

Is raised when a circuit algebra object cannot be converted to a concrete ABCD object.

exception qnet.algebra.circuit_algebra.**CannotVisualize**
Bases: *qnet.algebra.abstract_algebra.AlgebraException*

Is raised when a circuit algebra object cannot be visually represented.

exception qnet.algebra.circuit_algebra.**WrongCDimError**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Is raised when two object are tried to joined together in series but have different channel dimensions.

exception qnet.algebra.circuit_algebra.**IncompatibleBlockStructures**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Is raised when a circuit decomposition into a block-structure is requested that is icompatible with the actual block structure of the circuit expression.

exception qnet.algebra.circuit_algebra.**CannotEliminateAutomatically**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Raised when attempted automatic adiabatic elimination fails.

qnet.algebra.circuit_algebra.**check_cdims**(*cls*, *ops*, *kwargs*)
Check that all operands (*ops*) have equal channel dimension.

class qnet.algebra.circuit_algebra.**Circuit**
Bases: object

Abstract base class for the circuit algebra elements.

cdim
The channel dimension of the circuit expression, i.e. the number of external bosonic noises/inputs that the circuit couples to.

block_structure
If the circuit is *reducible* (i.e., it can be represented as a :py:class:Concatenation: of individual circuit expressions), this gives a tuple of cdim values of the subblocks. E.g. if A and B are irreducible and have `A.cdim = 2`, `B.cdim = 3`

```
>>> A = CircuitSymbol('A', 2)
>>> B = CircuitSymbol('B', 3)
```

Then the block structure of their Concatenation is:

```
>>> (A + B).block_structure
(2, 3)
```

while

```
>>> A.block_structure
(2,)
```

```
>>> B.block_structure
(3,)
```

index_in_block (*channel_index: int*) → int

Return the index a channel has within the subblock it belongs to. I.e., only for reducible circuits, this gives a result different from the argument itself.

Parameters **channel_index** (*int*) – The index of the external channel

Raises ValueError – for an invalid *channel_index*

get_blocks (*block_structure=None*)

For a reducible circuit, get a sequence of subblocks that when concatenated again yield the original circuit. The block structure given has to be compatible with the circuits actual block structure, i.e. it can only be more coarse-grained.

Parameters **block_structure** (*tuple*) – The block structure according to which the subblocks are generated (default = None, corresponds to the circuit's own block structure)

Returns A tuple of subblocks that the circuit consists of.

Raises *IncompatibleBlockStructures*

series_inverse () → qnet.algebra.circuit_algebra.Circuit

Return the inverse object (under the series product) for a circuit. In general for any X

```
>>> X = CircuitSymbol('X', cdim=3)
>>> (X << X.series_inverse() == X.series_inverse() << X ==
...  cid(X.cdim))
True
```

feedback (**, out_port=None, in_port=None*)

Return a circuit with self-feedback from the output port (zero-based) `out_port` to the input port `in_port`.

Parameters

- **out_port** (*int or NoneType*) – The output port from which the feedback connection leaves (zero-based, default = None corresponds to the *last* port).
- **in_port** (*int or NoneType*) – The input port into which the feedback connection goes (zero-based, default = None corresponds to the *last* port).

show ()

Show the circuit expression in an IPython notebook.

render (*fname=''*)

Render the circuit expression and store the result in a file

Parameters **fname** (*str*) – Path to an image file to store the result in.

Return (str) The path to the image file

creduce () → qnet.algebra.circuit_algebra.Circuit

If the circuit is reducible, try to reduce each subcomponent once. Depending on whether the components at the next hierarchy-level are themselves reducible, successive `circuit.creduce()` operations yields an increasingly fine-grained decomposition of a circuit into its most primitive elements.

toSLH () → qnet.algebra.circuit_algebra.SLH

Return the SLH representation of a circuit. This can fail if there are un-substituted pure circuit all_symbols

(`CircuitSymbol`) left in the expression or if the circuit includes *non-passive* ABCD models (cf.[1])

toABCD (*linearize=False*) → qnet.algebra.circuit_algebra.ABCD
Return the ABCD representation of a circuit expression. If *linearize=True* all operator expressions giving rise to non-linear equations of motion are dropped. This can fail if there are un-substituted pure circuit all_symbols (`CircuitSymbol`) left in the expression or if *linearize = False* and the circuit includes non-linear SLH models. (cf.[1])

coherent_input (**input_amps*) → qnet.algebra.circuit_algebra.Circuit
Feed coherent input amplitudes into the circuit. E.g. For a circuit with channel dimension of two, *C.coherent_input(0,1)* leads to an input amplitude of zero into the first and one into the second port.

Parameters **input_amps** (*SCALAR_TYPES*) – The coherent input amplitude for each port

Returns The circuit including the coherent inputs.

Return type *Circuit*

Raise WrongCDimError

space
Hilbert space of the circuit

class qnet.algebra.circuit_algebra.**SLH** (*S*, *L*, *H*)
Bases: `qnet.algebra.circuit_algebra.Circuit`, `qnet.algebra.abstract_algebra.Expression`

SLH class to encapsulate an open system model that is parametrized as described in[2] ,[3]

```
SLH(S, L, H)
```

Attributes

- **S** (*Matrix*) – The scattering matrix (with in general Operator-valued elements)
- **L** (*Matrix*) – The coupling vector (with in general Operator-valued elements)
- **H** (*Operator*) – The internal Hamiltonian operator

Parameters

- **S** – Value for the *S* attribute.
- **L** – Value for the *L* attribute
- **H** – Value for the *H* attribute

args

Ls
Lindblad operators (entries of the L vector), as a list

cdim

space
Total Hilbert space

[1] Gough, James & Nurdin (2010). Squeezing components in linear quantum feedback networks. Physical Review A, 81(2). doi:10.1103/PhysRevA.81.023804

[2] Gough & James (2008). Quantum Feedback Networks: Hamiltonian Formulation. Communications in Mathematical Physics, 287(3), 1109-1132. doi:10.1007/s00220-008-0698-8

[3] Gough & James (2009). The Series Product and Its Application to Quantum Feedforward and Feedback Networks. IEEE Transactions on Automatic Control, 54(11), 2530-2544. doi:10.1109/TAC.2009.2031205

all_symbols()
Set of all symbols occcuring in S, L, or H

series_with_slh(*other*)
Evaluate the series product with another :py:class:`SLH` object.

Parameters **other** (`SLH`) – An upstream SLH circuit.

Returns The combines system.

Return type *SLH*

concatenate_slh(*other*)
Evaluate the concatenation product with another SLH object.

expand()
Expand out all operator expressions within S, L and H and return a new SLH object with these expanded expressions.

simplify_scalar()
Simplify all scalar expressions within S, L and H and return a new SLH object with the simplified expressions.

symbolic_liouvillian()

symbolic_master_equation(*rho=None*)
Compute the symbolic Liouvillian acting on a state rho. If no rho is given, an OperatorSymbol is created in its place. This correspnds to the RHS of the master equation in which an average is taken over the external noise degrees of freedom.

Parameters **rho** (`Operator`) – A symbolic density matrix operator

Returns The RHS of the master equation.

Return type *Operator*

symbolic_heisenberg_eom(*X=None*, *noises=None*, *expand_simplify=True*)
Compute the symbolic Heisenberg equations of motion of a system operator X. If no X is given, an OperatorSymbol is created in its place. If no noises are given, this correspnds to the ensemble-averaged Heisenberg equation of motion.

Parameters

- **X** (`Operator`) – A system operator
- **noises** (`Operator`) – A vector of noise inputs

Returns The RHS of the Heisenberg equations of motion of X.

Return type *Operator*

class qnet.algebra.circuit_algebra.**ABCD**(*A*, *B*, *C*, *D*, *w*, *space*)
Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Expression*

ABCD model class in amplitude representation.

```
ABCD(A, B, C, D, w, space)
```

I.e. for a doubled up vector a = (a_1, ..., a_n, a_1^*, ... a_n^*)^T = double_up((a_1, ..., a_n)^T) and doubled up noises dA = (dA_1, ..., dA_m, dA_1^*, ..., dA_m^*)^T = double_up((dA_1, ..., dA_n)^T) The equation of motion for a is

$$da = Aadt + B(dA + double_up(w)dt)$$

The output field dA' is given by

$$dA' = Cadt + D(dA + double_u p(w)dt)$$

Parameters

- **A** (Matrix) – Coupling matrix: internal to internal, scalar valued elements, `shape = (2*n,2*n)`
- **B** (Matrix) – Coupling matrix external input to internal, scalar valued elements, `shape = (2*n,2*m)`
- **C** (Matrix) – Coupling matrix internal to external output, scalar valued elements, `shape = (2*m,2*n)`
- **D** (Matrix) – Coupling matrix external input to output, scalar valued elements, `shape = (2*m,2*m)`
- **w** (Matrix) – Coherent input amplitude vector, **NOT DOUBLED UP**, scalar valued elements, `shape = (m,1)`
- **space** (HilbertSpace) – Hilbert space with exactly n local factor spaces corresponding to the n internal degrees of freedom.

space
Total Hilbert space

args

n
The number of oscillators (int).

m
The number of external fields (int)

cdim
Dimension of circuit

class qnet.algebra.circuit_algebra.**CircuitSymbol**(*name*, *cdim*)
Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Expression*

Circuit Symbol object, parametrized by an identifier (name) and channel dimension.

name

args

all_symbols()

cdim
Dimension of circuit

space
FullSpace (Circuit Symbols are not restricted to a particular Hilbert space)

class qnet.algebra.circuit_algebra.**CPermutation**(*permutation*)
Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Expression*

The channel permuting circuit. This circuit expression is only a rearrangement of input and output fields. A channel permutation is given as a tuple of image points. Permutations are usually represented as

A permutation $\sigma \in \Sigma_n$ of n elements is often represented in the following form

$$\begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix},$$

but obviously it is fully sufficient to specify the tuple of images $(\sigma(1), \sigma(2), \dots, \sigma(n))$. We thus parametrize our permutation circuits only in terms of the image tuple. Moreover, we will be working with *zero-based indices*!

A channel permutation circuit for a given permutation (represented as a python tuple of image indices) scatters the j-th input field to the $\sigma(j)$-th output field.

It is instantiated as

```
CPermutation(permutation)
```

Parameters **permutation** (*tuple*) – Channel permutation image tuple.

classmethod create (*permutation*)

args

block_perms

If the circuit is reducible into permutations within subranges of the full range of channels, this yields a tuple with the internal permutations for each such block.

Type tuple

permutation

The permutation image tuple.

cdim

series_with_permutation (*other*)

Compute the series product with another channel permutation circuit.

Returns The composite permutation circuit (could also be the identity circuit for n channels)

Return type *Circuit*

space

TrivialSpace

class qnet.algebra.circuit_algebra.**SeriesProduct** (**operands*, ***kwargs*)

Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Operation*

The series product circuit operation. It can be applied to any sequence of circuit objects that have equal channel dimension.

```
SeriesProduct(*operands)
```

Parameters **operands** – Circuits in feedforward configuration.

neutral_element = neutral_element

cdim

space

Hilbert space of the series product (product space of all operators)

class qnet.algebra.circuit_algebra.**Concatenation** (**operands*)

Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Operation*

The concatenation product circuit operation. It can be applied to any sequence of circuit objects.

`Concatenation(*operands)`

Parameters **operands** (Circuit) – Circuits in parallel configuration.

neutral_element = CircuitZero

cdim
Circuit dimension (sum of dimensions of the operands)

space
Hilbert space of the Concatenation (Product space of all operators)

class qnet.algebra.circuit_algebra.**Feedback**(*circuit: qnet.algebra.circuit_algebra.Circuit*, *, *out_port: int*, *in_port: int*)
Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Operation*

The circuit feedback operation applied to a circuit of channel dimension > 1 and an from an output port index to an input port index.

Parameters

- **circuit** (Circuit) – The circuit that undergoes self-feedback
- **out_port** (*int*) – The output port index.
- **in_port** (*int*) – The input port index.

delegate_to_method = (<class 'qnet.algebra.circuit_algebra.Concatenation'>, <class 'qnet.algebra.circuit_algebra.SI

kwargs

operand
The Circuit that undergoes feedback

out_in_pair
Tuple of zero-based feedback port indices (out_port, in_port)

cdim
Circuit dimension (one less than the circuit on which the feedback acts

classmethod create(*circuit: qnet.algebra.circuit_algebra.Circuit*, *, *out_port: int*, *in_port: int*) → qnet.algebra.circuit_algebra.Feedback

space
Hilbert space of the Feedback circuit (same as the Hilbert space of the operand)

class qnet.algebra.circuit_algebra.**SeriesInverse**(**operands*, ***kwargs*)
Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Operation*

Symbolic series product inversion operation.

`SeriesInverse(circuit)`

One generally has

```
>>> C = CircuitSymbol('C', cdim=3)
>>> SeriesInverse(C) << C == cid(C.cdim)
True
```

and

```
>>> C << SeriesInverse(C) == cid(C.cdim)
True
```

Parameters **circuit** (`Circuit`) – The circuit system to invert.

delegate_to_method = (<class 'qnet.algebra.circuit_algebra.SeriesProduct'>, <class 'qnet.algebra.circuit_algebra.Co

operand
The un-inverted circuit

classmethod create (*circuit*)

cdim

space
Hilbert space of the series inversion circuit (same Hilbert space as the series product being inverted)

qnet.algebra.circuit_algebra.**circuit_identity** (*n*)
Return the circuit identity for n channels.

Parameters **n** (*int*) – The channel dimension

Returns n-channel identity circuit

Return type *Circuit*

qnet.algebra.circuit_algebra.**cid** (*n*)
Return the circuit identity for n channels.

Parameters **n** (*int*) – The channel dimension

Returns n-channel identity circuit

Return type *Circuit*

qnet.algebra.circuit_algebra.**P_sigma** (**permutation*)
Create a channel permutation circuit for the given index image values. :param permutation: image points :type permutation: int :return: CPermutation.create(permutation) :rtype: Circuit

qnet.algebra.circuit_algebra.**FB** (*circuit*, *, *out_port=None*, *in_port=None*)
Wrapper for :py:class:Feedback: but with additional default values.

Parameters

- **circuit** (`Circuit`) – The circuit that undergoes self-feedback
- **out_port** (*int*) – The output port index, default = None –> last port
- **in_port** (*int*) – The input port index, default = None –> last port

Returns The circuit with applied feedback operation.

Return type *Circuit*

qnet.algebra.circuit_algebra.**get_common_block_structure** (*lhs_bs*, *rhs_bs*)
For two block structures `aa = (a1, a2, ..., an)`, `bb = (b1, b2, ..., bm)` generate the maximal common block structure so that every block from aa and bb is contained in exactly one block of the resulting structure. This is useful for determining how to apply the distributive law when feeding two concatenated Circuit objects into each other.

Examples

(1, 1, 1), (2, 1) -> (2, 1) (1, 1, 2, 1), (2, 1, 2) -> (2, 3)

Parameters

- **lhs_bs** (*tuple*) – first block structure
- **rhs_bs** (*tuple*) – second block structure

qnet.algebra.circuit_algebra.**extract_signal**(*k*, *n*)

Create a permutation that maps the k-th (zero-based) element to the last element, while preserving the relative order of all other elements.

Parameters

- **k** (*int*) – The index to extract
- **n** (*int*) – The total number of elements

Returns Permutation image tuple

Return type tuple

qnet.algebra.circuit_algebra.**extract_signal_circuit**(*k*, *cdim*)

Create a channel permutation circuit that maps the k-th (zero-based) input to the last output, while preserving the relative order of all other channels.

Parameters

- **k** (*int*) – Extracted channel index
- **cdim** (*int*) – The channel dimension

Returns Permutation circuit

Return type *Circuit*

qnet.algebra.circuit_algebra.**map_signals**(*mapping*, *n*)

For a given {input:output} mapping in form of a dictionary, generate the permutation that achieves the specified mapping while leaving the relative order of all non-specified elements intact. :param mapping: Input-output mapping of indices (zero-based) {in1:out1, in2:out2,...} :type mapping: dict :param n: total number of elements :type n: int :return: Signal mapping permutation image tuple :rtype: tuple :raise: ValueError

qnet.algebra.circuit_algebra.**map_signals_circuit**(*mapping*, *n*)

For a given {input:output} mapping in form of a dictionary, generate the channel permutating circuit that achieves the specified mapping while leaving the relative order of all non-specified channels intact. :param mapping: Input-output mapping of indices (zero-based) {in1:out1, in2:out2,...} :type mapping: dict :param n: total number of elements :type n: int :return: Signal mapping permutation image tuple :rtype: Circuit

qnet.algebra.circuit_algebra.**pad_with_identity**(*circuit*, *k*, *n*)

Pad a circuit by 'inserting' an n-channel identity circuit at index k. I.e., a circuit of channel dimension N is extended to one of channel dimension N+n, where the channels k, k+1, ...k+n-1, just pass through the system unaffected. E.g. let A, B be two single channel systems

```
>>> A = CircuitSymbol('A', 1)
>>> B = CircuitSymbol('B', 1)
>>> print(ascii(pad_with_identity(A+B, 1, 2)))
A + cid(2) + B
```

This method can also be applied to irreducible systems, but in that case the result can not be decomposed as nicely.

Parameters

- **k** (*int*) – The index at which to insert the circuit
- **n** (*int*) – The number of channels to pass through

Returns An extended circuit that passes through the channels k, k+1, ..., k+n-1

Return type *Circuit*

`qnet.algebra.circuit_algebra.`**`getABCD`**(*slh*, *a0=None*, *doubled_up=True*)

Return the A, B, C, D and (a, c) matrices that linearize an SLH model about a coherent displacement amplitude a0.

The equations of motion and the input-output relation are then:

dX = (A X + a) dt + B dA_in dA_out = (C X + c) dt + D dA_in

where, if doubled_up == False

dX = [a_1, ..., a_m] dA_in = [dA_1, ..., dA_n]

or if doubled_up == True

dX = [a_1, ..., a_m, a_1^*, ... a_m^*] dA_in = [dA_1, ..., dA_n, dA_1^*, ..., dA_n^*]

Parameters

- **slh** – SLH object
- **a0** – dictionary of coherent amplitudes {a1: a1_0, a2: a2_0, ...} with annihilation mode operators as keys and (numeric or symbolic) amplitude as values.
- **doubled_up** – boolean, necessary for phase-sensitive / active systems

Returns SymPy matrix objects :returns: A tuple (A, B, C, D, a, c])

A: coupling of modes to each other B: coupling of external input fields to modes C: coupling of internal modes to output D: coupling of external input fields to output fields

a: constant coherent input vector for mode e.o.m. c: constant coherent input vector of scattered amplitudes contributing to the output

`qnet.algebra.circuit_algebra.`**`move_drive_to_H`**(*slh*, *which=[]*)

For the given *slh* model, move inhomogeneities in the Lindblad operators (resulting from the presence of a coherent drive, see `Displace`) to the Hamiltonian.

This exploits the invariance of the Lindblad master equation under the transformation (cf. Breuer and Pettrucione, Ch 3.2.1)

$$L_i \longrightarrow L_i' = L_i - \alpha_i \tag{10.1}$$

$$H \longrightarrow H' = H + \frac{1}{2i} \sum_j (\alpha_j L_j^\dagger - \alpha_j^* L_j) \tag{10.2}$$

In the context of SLH, this transformation is achieved by feeding *slh* into

$$(, -\alpha, 0)$$

where α has the components α_i.

The *which* argument allows to select which subscripts i (circuit dimensions) should be tranformed. The default is all dimensions. If *slh* does not contain any inhomogeneities, it is invariant under the transformation.

`qnet.algebra.circuit_algebra.`**`prepare_adiabatic_limit`**(*slh*, *k=None*)

Prepare the adiabatic elimination procedure for an SLH object with scaling parameter k->infty

Parameters

- **slh** – The SLH object to take the limit for
- **k** – The scaling parameter.

Returns The objects Y, A, B, F, G, N necessary to compute the limiting system.

Return type tuple

qnet.algebra.circuit_algebra.**eval_adiabatic_limit**(*YABFGN*, *Ytilde*, *P0*)

Compute the limiting SLH model for the adiabatic approximation.

Parameters

- **YABFGN** – The tuple (Y, A, B, F, G, N) as returned by prepare_adiabatic_limit.
- **Ytilde** – The pseudo-inverse of Y, satisfying Y * Ytilde = P0.
- **P0** – The projector onto the null-space of Y.

Returns Limiting SLH model

Return type *SLH*

qnet.algebra.circuit_algebra.**try_adiabatic_elimination**(*slh*, *k=None*, *fock_trunc=6*, *sub_P0=True*)

Attempt to automatically carry out the adiabatic elimination procedure on slh with scaling parameter k.

This will project the Y operator onto a truncated basis with dimension specified by fock_trunc. sub_P0 controls whether an attempt is made to replace the kernel projector P0 by an IdentityOperator.

qnet.algebra.circuit_algebra.**connect**(*components*, *connections*, *force_SLH=False*, *expand_simplify=True*)

Connect a list of components according to a list of connections.

Parameters

- **components** (*list*) – List of Circuit instances
- **connections** (*list*) – List of pairs `((c1, port1), (c2, port2))` where `c1` and `c2` are elements of *components* (or the index of the element in *components*, and `port1` and `port2` are the indices of the ports of the two components that should be connected
- **force_SLH** (*bool*) – If True, convert the result to an SLH object
- **expand_simplify** (*bool*) – If the result is an SLH object, expand and simplify the circuit after each feedback connection is added

qnet.algebra.hilbert_space_algebra module

This module defines some simple classes to describe simple and *composite/tensor* (i.e., multiple degree of freedom) Hilbert spaces of quantum systems.

For more details see *Hilbert Space Algebra*.

Summary

Exceptions:

BasisNotSetError	Raised if the basis or a Hilbert space dimension is requested but is not

Classes:

HilbertSpace	Basic Hilbert space class from which concrete classes are derived.
LocalSpace	A local Hilbert space, i.e., for a single degree of freedom.
ProductSpace	Tensor product space class for an arbitrary number of LocalSpace factors.

Functions:

convert_to_spaces	For all operands that are merely of type str or int, substitute LocalSpace objects with corresponding labels: For a string, just itself, for an int, a string version of that int.
empty_trivial	A ProductSpace of zero Hilbert spaces should yield the TrivialSpace

`__all__`: *BasisNotSetError*, *FullSpace*, *HilbertSpace*, *LocalSpace*, *ProductSpace*, *TrivialSpace*

Module data:

qnet.algebra.hilbert_space_algebra.**FullSpace**

qnet.algebra.hilbert_space_algebra.**TrivialSpace**

Reference

exception qnet.algebra.hilbert_space_algebra.**BasisNotSetError**

Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Raised if the basis or a Hilbert space dimension is requested but is not available

qnet.algebra.hilbert_space_algebra.**convert_to_spaces**(*cls*, *ops*, *kwargs*)

For all operands that are merely of type str or int, substitute LocalSpace objects with corresponding labels: For a string, just itself, for an int, a string version of that int.

qnet.algebra.hilbert_space_algebra.**empty_trivial**(*cls*, *ops*, *kwargs*)

A ProductSpace of zero Hilbert spaces should yield the TrivialSpace

class qnet.algebra.hilbert_space_algebra.**HilbertSpace**

Bases: object

Basic Hilbert space class from which concrete classes are derived.

tensor(**others*)

Tensor product between Hilbert spaces

Parameters **others** (HilbertSpace) – Other Hilbert space(s)

Returns Tensor product space.

Return type *HilbertSpace*

remove(*other*)

Remove a particular factor from a tensor product space.

intersect(*other*)

Find the mutual tensor factors of two Hilbert spaces.

local_factors
Return tuple of LocalSpace objects that tensored together yield this Hilbert space.

isdisjoint(*other*)
Check whether two Hilbert spaces are disjoint (do not have any common local factors). Note that *FullSpace* is *not* disjoint with any other Hilbert space, while *TrivialSpace is* disjoint with any other HilbertSpace (even itself)

is_tensor_factor_of(*other*)
Test if a space is included within a larger tensor product space. Also `True` if `self == other`.

Parameters **other** (`HilbertSpace`) – Other Hilbert space

Return type bool

is_strict_tensor_factor_of(*other*)
Test if a space is included within a larger tensor product space. Not `True` if `self == other`.

dimension
The full dimension of the Hilbert space (or None) if the dimension is not known

get_dimension(*raise_basis_not_set_error=True*)
Return the *dimension* property, but if *raise_basis_not_set_error* is True, raise a *BasisNotSetError* if no basis is set, instead of returning None

basis
Basis of the the Hilbert space, or None if no basis is set

get_basis(*raise_basis_not_set_error=True*)
Return the *basis* property, but if *raise_basis_not_set_error* is True, raise a *BasisNotSetError* if no basis is set, instead of returning None

is_strict_subfactor_of(*other*)
Test whether a Hilbert space occures as a strict sub-factor in (larger) Hilbert space

__len__()
The number of LocalSpace factors / degrees of freedom.

class qnet.algebra.hilbert_space_algebra.**LocalSpace**(*label*, *, *basis=None*, *dimension=None*, *order_index=None*)
Bases: *qnet.algebra.hilbert_space_algebra.HilbertSpace*, *qnet.algebra.abstract_algebra.Expression*

A local Hilbert space, i.e., for a single degree of freedom.

Parameters

- **label** (*str*) – label (subscript) of the Hilbert space
- **basis** (*tuple or None*) – Set an explicit basis for the Hilbert space (tuple of labels for the basis states)
- **dimension** (*int or None*) – Specify the dimension n of the Hilbert space. This implies a basis numbered from 0 to $n-1$.
- **order_index** (*int or None*) – An optional key that determines the preferred order of Hilbert spaces. This also changes the order of e.g. sums or products of Operators. Hilbert spaces will be ordered from left to right be increasing *order_index*; Hilbert spaces without an explicit *order_index* are sorted by their label

args

label
Label of the Hilbert space

basis

dimension

kwargs

minimal_kwargs

all_symbols()

remove(*other*)

intersect(*other*)

local_factors

is_strict_subfactor_of(*other*)

class qnet.algebra.hilbert_space_algebra.**ProductSpace**(**local_spaces*)
Bases: *qnet.algebra.hilbert_space_algebra.HilbertSpace*, *qnet.algebra.abstract_algebra.Operation*

Tensor product space class for an arbitrary number of LocalSpace factors.

```
>>> hs1 = LocalSpace('1', basis=(0,1))
>>> hs2 = LocalSpace('2', basis=(0,1))
>>> hs = hs1 * hs2
>>> hs.basis
('0,0', '0,1', '1,0', '1,1')
```

signature = ((<class 'qnet.algebra.hilbert_space_algebra.HilbertSpace'>, '*'), {})

neutral_element = TrivialSpace

classmethod create(**local_spaces*)

basis
Basis of the ProductSpace, from the bases of the operands

dimension

remove(*other*)
Remove a particular factor from a tensor product space.

local_factors
The LocalSpace instances that make up the product

classmethod order_key(*obj*)
Key by which operands are sorted

intersect(*other*)
Find the mutual tensor factors of two Hilbert spaces.

is_strict_subfactor_of(*other*)
Test if a space is included within a larger tensor product space. Not `True` if `self == other`.

qnet.algebra.matrix_algebra module

Matrices of Operators

Summary

Exceptions:

NonSquareMatrix	

Classes:

Matrix	Matrix with Operator (or scalar-) valued elements.

Functions:

Im	The imaginary part of a number or operator.
ImAdjoint	The imaginary part of an OperatorMatrix, i.e.
ImMatrix	
Re	The real part of a number or operator.
ReAdjoint	The real part of an OperatorMatrix, i.e.
ReMatrix	
block_matrix	Generate the operator matrix with quadrants
diagm	Generalizes the diagonal matrix creation capabilities of *numpy.diag* to OperatorMatrix objects.
hstackm	Generalizes *numpy.hstack* to OperatorMatrix objects.
identity_matrix	Generate the N-dimensional identity matrix.
permutation_matrix	Return an orthogonal permutation matrix
vstackm	Generalizes *numpy.vstack* to OperatorMatrix objects.
zerosm	Generalizes `numpy.zeros` to *Matrix* objects.

`__all__`: *ImAdjoint*, *ImMatrix*, *Matrix*, *NonSquareMatrix*, *ReAdjoint*, *ReMatrix*, *block_matrix*, *diagm*, *hstackm*, *identity_matrix*, *permutation_matrix*, *vstackm*, *zerosm*

Reference

exception `qnet.algebra.matrix_algebra.`**`NonSquareMatrix`**

Bases: `Exception`

class `qnet.algebra.matrix_algebra.`**`Matrix`**(*m*)

Bases: *qnet.algebra.abstract_algebra.Expression*

Matrix with Operator (or scalar-) valued elements.

`matrix` **= None**

`shape`

The shape of the matrix `(nrows, ncols)`

`block_structure`

For square matrices this gives the block (-diagonal) structure of the matrix as a tuple of integers that sum up to the full dimension.

Type tuple

`args`

`is_zero`

Are all elements of the matrix zero?

transpose()
The transpose matrix

conjugate()
The element-wise conjugate matrix, i.e., if an element is an operator this means the adjoint operator, but no transposition of matrix elements takes place.

T
Transpose matrix

adjoint()
Return the adjoint operator matrix, i.e. transpose and the Hermitian adjoint operators of all elements.

dag()
Return the adjoint operator matrix, i.e. transpose and the Hermitian adjoint operators of all elements.

trace()

H
Return the adjoint operator matrix, i.e. transpose and the Hermitian adjoint operators of all elements.

element_wise(*method*)
Apply a method to each matrix element and return the result in a new operator matrix of the same shape. :param method: A method taking a single argument. :type method: FunctionType :return: Operator matrix with results of method applied element-wise. :rtype: Matrix

series_expand(*param*, *about*, *order*)
Expand the matrix expression as a truncated power series in a scalar parameter.

Parameters

- **param** (*sympy.core.symbol.Symbol*) – Expansion parameter.
- **about** (*Any one of Operator.scalar_types*) – Point about which to expand.
- **order** (*int >= 0*) – Maximum order of expansion.

Returns tuple of length (order+1), where the entries are the expansion coefficients.

Return type tuple of Operator

expand()
Expand each matrix element distributively. :return: Expanded matrix. :rtype: Matrix

all_symbols()

space
Combined Hilbert space of all matrix elements.

simplify_scalar()
Simplify all scalar expressions appearing in the Matrix.

qnet.algebra.matrix_algebra.**hstackm**(*matrices*)
Generalizes *numpy.hstack* to OperatorMatrix objects.

qnet.algebra.matrix_algebra.**vstackm**(*matrices*)
Generalizes *numpy.vstack* to OperatorMatrix objects.

qnet.algebra.matrix_algebra.**diagm**(*v*, *k=0*)
Generalizes the diagonal matrix creation capabilities of *numpy.diag* to OperatorMatrix objects.

qnet.algebra.matrix_algebra.**block_matrix**(*A*, *B*, *C*, *D*)
Generate the operator matrix with quadrants

$$\begin{pmatrix} AB \\ CD \end{pmatrix}$$

Parameters

- **A** (Matrix) – Matrix of shape `(n, m)`
- **B** (Matrix) – Matrix of shape `(n, k)`
- **C** (Matrix) – Matrix of shape `(l, m)`
- **D** (Matrix) – Matrix of shape `(l, k)`

Returns The combined block matrix [[A, B], [C, D]].

Type OperatorMatrix

`qnet.algebra.matrix_algebra.`**`identity_matrix`**`(`*N*`)`

Generate the N-dimensional identity matrix.

Parameters N (*int*) – Dimension

Returns Identity matrix in N dimensions

Return type *Matrix*

`qnet.algebra.matrix_algebra.`**`zerosm`**`(`*shape*, **args*, ***kwargs*`)`

Generalizes `numpy.zeros` to *Matrix* objects.

`qnet.algebra.matrix_algebra.`**`permutation_matrix`**`(`*permutation*`)`

Return an orthogonal permutation matrix M_σ for a permutation σ defined by the image tuple $(\sigma(1), \sigma(2), \dots \sigma(n))$, such that

$$M_\sigma \vec{e}_i = \vec{e}_{\sigma(i)}$$

where $\vec{e}_k$ is the k-th standard basis vector. This definition ensures a composition law:

$$M_{\sigma \cdot \tau} = M_\sigma M_\tau.$$

The column form of M_σ is thus given by

$$M = (\vec{e}_{\sigma(1)}, \vec{e}_{\sigma(2)}, \dots \vec{e}_{\sigma(n)}).$$

Parameters permutation (*tuple*) – A permutation image tuple (zero-based indices!)

`qnet.algebra.matrix_algebra.`**`Im`**`(`*op*`)`

The imaginary part of a number or operator. Acting on OperatorMatrices, it produces the element-wise imaginary parts.

Parameters op (*Operator or Matrix or any of Operator.scalar_types*) – Anything that has a conjugate method.

Returns The imaginary part of the operand.

Return type Same as type of *op*.

`qnet.algebra.matrix_algebra.`**`Re`**`(`*op*`)`

The real part of a number or operator. Acting on OperatorMatrices, it produces the element-wise real parts.

Parameters op (*Operator or Matrix or any of Operator.scalar_types*) – Anything that has a conjugate method.

Returns The real part of the operand.

Return type Same as type of *op*.

`qnet.algebra.matrix_algebra.`**`ImAdjoint`**`(`*opmatrix*`)`

The imaginary part of an OperatorMatrix, i.e. a Hermitian OperatorMatrix :param opmatrix: The operand. :type opmatrix: Matrix :return: The matrix imaginary part of the operand. :rtype: Matrix

`qnet.algebra.matrix_algebra.`**`ReAdjoint`**(*opmatrix*)
The real part of an OperatorMatrix, i.e. a Hermitian OperatorMatrix :param opmatrix: The operand. :type opmatrix: Matrix :return: The matrix real part of the operand. :rtype: Matrix

qnet.algebra.operator_algebra module

This module features classes and functions to define and manipulate symbolic Operator expressions. For more details see *The Operator Algebra module*.

For a list of all properties and methods of an operator object, see the documentation for the basic `Operator` class.

Summary

Classes:

`Adjoint`	The symbolic Adjoint of an operator.
`Create`	`Create(hs=space)` yields a bosonic creation operator acting on a
`Destroy`	`Destroy(hs=space)` yields a bosonic annihilation operator acting on a
`Displace`	Unitary coherent displacement operator
`Jminus`	`Jminus(space)` yields the J_- lowering ladder operator of a general
`Jplus`	`Jplus(space)` yields the J_+ raising ladder operator of a general
`Jz`	`Jz(space)` yields the z component of a general spin operator acting
`LocalOperator`	Base class for all kinds of operators that act *locally*, i.e.
`LocalSigma`	A local level flip operator operator acting on a particular local space/degree of freedom.
`NullSpaceProjector`	Returns a projection operator $\mathcal{P}_{{\rm Ker} X}$ that
`Operator`	The basic operator class, which fixes the abstract interface of operator objects and where possible also defines the default behavior under operations.
`OperatorOperation`	Base class for Operations acting only on Operator arguments, for when the Hilbert space of the operation result is the product space of the operands.
`OperatorPlus`	A sum of Operators
`OperatorPlusMinusCC`	An operator plus or minus its complex conjugate
`OperatorSymbol`	Symbolic operator, parametrized by an identifier string and an associated Hilbert space.
`OperatorTimes`	A product of Operators that serves both as a product within a Hilbert space as well as a tensor product.
`OperatorTrace`	Take the (partial) trace of an operator X over the degrees of
`Phase`	The unitary Phase operator acting on a particular local space/degree of
`PseudoInverse`	The symbolic pseudo-inverse X^+ of an operator X.
`ScalarTimesOperator`	Multiply an operator by a scalar coefficient.
`SingleOperatorOperation`	Base class for Operations that act on a single Operator

Continued on next page

Table 10.13 – continued from previous page

Squeeze	A unitary Squeezing operator acting on a particular local space/degree

Functions:

Jmjmcoeff	
Jpjmcoeff	
Jzjmcoeff	
LocalProjector	
X	Pauli-type X-operator
Y	Pauli-type Y-operator
Z	Pauli-type Z-operator
adjoint	Return the adjoint of an obj.
create_operator_pm_cc	Return a list of rules that can be used in an
decompose_space	Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated partial traces.
delegate_to_method	Create a simplification rule that delegates the instantiation to the
expand_operator_pm_cc	Return a list of rules that can be used in *simplify* to expand
factor_coeff	Factor out coefficients of all factors.
factor_for_trace	Given a local space ls to take the partial trace over and an operator, factor the trace such that operators acting on disjoint degrees of freedom are pulled out of the trace.
get_coeffs	Create a dictionary with all Operator terms of the expression (understood as a sum) as keys and their coefficients as values.
implied_local_space	Return a simplification that converts the positional argument
scalar_free_symbols	Return all free symbols from any symbolic operand
simplify_scalar	Simplify all occurences of scalar expressions in s
space	Gives the associated HilbertSpace with an object.

`__all__`: *Adjoint*, *Create*, *Destroy*, *Displace*, *II*, *IdentityOperator*, *Jminus*, *Jmjmcoeff*, *Jpjmcoeff*, *Jplus*, *Jz*, *Jzjmcoeff*, *LocalOperator*, *LocalProjector*, *LocalSigma*, *NullSpaceProjector*, *Operator*, *OperatorOperation*, *OperatorPlus*, *OperatorPlusMinusCC*, *OperatorSymbol*, *OperatorTimes*, *OperatorTrace*, *Phase*, *PseudoInverse*, *ScalarTimesOperator*, *SingleOperatorOperation*, *Squeeze*, *X*, *Y*, *Z*, *ZeroOperator*, *adjoint*, *create_operator_pm_cc*, *decompose_space*, *expand_operator_pm_cc*, *factor_coeff*, *factor_for_trace*, *get_coeffs*, *scalar_free_symbols*, *simplify_scalar*, *space*

Module data:

qnet.algebra.operator_algebra.**II**

qnet.algebra.operator_algebra.**IdentityOperator**

qnet.algebra.operator_algebra.**ZeroOperator**

Reference

qnet.algebra.operator_algebra.**implied_local_space**(*, *arg_index=None*, *keys=None*)

Return a simplification that converts the positional argument *arg_index* from (str, int) to LocalSpace, as well as

any keyword argument with one of the given keys

qnet.algebra.operator_algebra.**delegate_to_method**(*mtd*)
Create a simplification rule that delegates the instantiation to the method *mtd* of the operand (if defined)

class qnet.algebra.operator_algebra.**Operator**
Bases: object

The basic operator class, which fixes the abstract interface of operator objects and where possible also defines the default behavior under operations. Any operator contains an associated HilbertSpace object, on which it is taken to act non-trivially.

space
The HilbertSpace on which the operator acts non-trivially

adjoint()
The Hermitian adjoint of the operator.

conjugate()
The Hermitian adjoint of the operator.

dag()
The Hermitian adjoint of the operator.

pseudo_inverse()
The pseudo-Inverse of the Operator, i.e., it inverts the operator on the orthogonal complement of its nullspace

expand()
Expand out distributively all products of sums. Note that this does not expand out sums of scalar coefficients.

Returns A fully expanded sum of operators.

Return type *Operator*

simplify_scalar()
Simplify all scalar coefficients within the Operator expression.

Returns The simplified expression.

Return type *Operator*

diff(*sym*, *n=1*, *expand_simplify=True*)
Differentiate by scalar parameter sym.

Parameters

- **sym** (*(sympy.Symbol)*) – What to differentiate by.
- **n** (*(int)*) – How often to differentiate
- **expand_simplify** (*(bool)*) – Whether to simplify the result.

Return (Operator) The n-th derivative.

series_expand(*param*, *about*, *order*)
Expand the operator expression as a truncated power series in a scalar parameter.

Parameters

- **param** (*sympy.core.symbol.Symbol*) – Expansion parameter.
- **about** (*Any one of SCALAR_TYPES*) – Point about which to expand.
- **order** (*int >= 0*) – Maximum order of expansion.

Returns tuple of length (order+1), where the entries are the expansion coefficients.

Return type tuple of Operator

class qnet.algebra.operator_algebra.**LocalOperator**(**args*, *hs*, *identifier=None*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Expression*

Base class for all kinds of operators that act *locally*, i.e. only on a single degree of freedom.

space

args

kwargs

minimal_kwargs

identifier
The name / identifying symbol of the operator

all_symbols()

class qnet.algebra.operator_algebra.**OperatorOperation**(**operands*, ***kwargs*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

Base class for Operations acting only on Operator arguments, for when the Hilbert space of the operation result is the product space of the operands.

space

class qnet.algebra.operator_algebra.**SingleOperatorOperation**(*op*, ***kwargs*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

Base class for Operations that act on a single Operator

space

operand

class qnet.algebra.operator_algebra.**OperatorSymbol**(*identifier*, ***, *hs*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Expression*

Symbolic operator, parametrized by an identifier string and an associated Hilbert space.

Parameters

- **identifier** (*str*) – Symbol identifier
- **hs** (HilbertSpace) – Associated Hilbert space (can be a ProductSpace)

args

kwargs

space

all_symbols()

class qnet.algebra.operator_algebra.**Create**(**args*, *hs*, *identifier=None*)
Bases: *qnet.algebra.operator_algebra.LocalOperator*

`Create(hs=space)` yields a bosonic creation operator acting on a particular local space/degree of freedom. Its adjoint is:

```
>>> print(ascii(Create(hs=1).adjoint()))
a^(1)
```

and it obeys the bosonic commutation relation:

```
>>> Destroy(hs=1) * Create(hs=1) - Create(hs=1) * Destroy(hs=1)
IdentityOperator
>>> Destroy(hs=1) * Create(hs=2) - Create(hs=2) * Destroy(hs=1)
ZeroOperator
```

Parameters **space** (LocalSpace or str) – Associated local Hilbert space.

class qnet.algebra.operator_algebra.**Destroy**(**args, hs, identifier=None*)

Bases: *qnet.algebra.operator_algebra.LocalOperator*

`Destroy(hs=space)` yields a bosonic annihilation operator acting on a particular local space/degree of freedom. Its adjoint is:

```
>>> print(ascii(Destroy(hs=1).adjoint()))
a^(1)H
```

and it obeys the bosonic commutation relation:

```
>>> Destroy(hs=1) * Create(hs=1) - Create(hs=1) * Destroy(hs=1)
IdentityOperator
>>> Destroy(hs=1) * Create(hs=2) - Create(hs=2) * Destroy(hs=1)
ZeroOperator
```

Parameters **space** (LocalSpace or str) – Associated local Hilbert space.

class qnet.algebra.operator_algebra.**Jz**(**args, hs, identifier=None*)

Bases: *qnet.algebra.operator_algebra.LocalOperator*

`Jz(space)` yields the z component of a general spin operator acting on a particular local space/degree of freedom with well defined spin quantum number J. It is Hermitian:

```
>>> print(ascii(Jz(hs=1).adjoint()))
J_z^(1)
```

Jz, *Jplus* and *Jminus* satisfy the angular momentum commutator algebra:

```
>>> print(ascii((Jz(hs=1) * Jplus(hs=1) -
...              Jplus(hs=1)*Jz(hs=1)).expand()))
J_+^(1)

>>> print(ascii((Jz(hs=1) * Jminus(hs=1) -
...              Jminus(hs=1)*Jz(hs=1)).expand()))
-J_-^(1)

>>> print(ascii((Jplus(hs=1) * Jminus(hs=1)
...              - Jminus(hs=1)*Jplus(hs=1)).expand()))
2 * J_z^(1)
```

where `Jplus = Jx + i * Jy`, `Jminux= Jx - i * Jy`.

class qnet.algebra.operator_algebra.**Jplus**(**args, hs, identifier=None*)

Bases: *qnet.algebra.operator_algebra.LocalOperator*

`Jplus(space)` yields the J_+ raising ladder operator of a general spin operator acting on a particular local space/degree of freedom with well defined spin quantum number J. It's adjoint is the lowering operator:

```
>>> print(ascii(Jplus(hs=1).adjoint()))
J_-^(1)
```

`Jz`, `Jplus` and `Jminus` satisfy that angular momentum commutator algebra:

```
>>> print(ascii((Jz(hs=1) * Jplus(hs=1) -
...              Jplus(hs=1)*Jz(hs=1)).expand()))
J_+^(1)

>>> print(ascii((Jz(hs=1) * Jminus(hs=1) -
...              Jminus(hs=1)*Jz(hs=1)).expand()))
-J_-^(1)

>>> print(ascii((Jplus(hs=1) * Jminus(hs=1) -
...             Jminus(hs=1)*Jplus(hs=1)).expand()))
2 * J_z^(1)
```

where `Jplus = Jx + i * Jy`, `Jminux= Jx - i * Jy`.

class `qnet.algebra.operator_algebra.`**`Jminus`**(**args*, *hs*, *identifier=None*)

Bases: `qnet.algebra.operator_algebra.LocalOperator`

`Jminus(space)` yields the J_- lowering ladder operator of a general spin operator acting on a particular local space/degree of freedom with well defined spin quantum number J. It's adjoint is the raising operator:

```
>>> print(ascii(Jminus(hs=1).adjoint()))
J_+^(1)
```

`Jz`, `Jplus` and `Jminus` satisfy that angular momentum commutator algebra:

```
>>> print(ascii((Jz(hs=1) * Jplus(hs=1) -
...              Jplus(hs=1)*Jz(hs=1)).expand()))
J_+^(1)

>>> print(ascii((Jz(hs=1) * Jminus(hs=1) -
...              Jminus(hs=1)*Jz(hs=1)).expand()))
-J_-^(1)

>>> print(ascii((Jplus(hs=1) * Jminus(hs=1) -
...              Jminus(hs=1)*Jplus(hs=1)).expand()))
2 * J_z^(1)
```

where `Jplus = Jx + i * Jy`, `Jminux= Jx - i * Jy`.

class `qnet.algebra.operator_algebra.`**`Phase`**(*phi*, ***, *hs*, *identifier=None*)

Bases: `qnet.algebra.operator_algebra.LocalOperator`

The unitary Phase operator acting on a particular local space/degree of freedom:

$$P_{\rm s}(\phi) := \exp\left(i\phi a_{\rm s}^\dagger a_{\rm s}\right)$$

where $a_{\rm s}$ is the annihilation operator acting on the local space s.

Parameters

- **hs** (`LocalSpace` *or* `str`) – Associated local Hilbert space.
- **phi** (Any from *SCALAR_TYPES*) – Displacement amplitude.

args

all_symbols()

class qnet.algebra.operator_algebra.**Displace**(*alpha*, ***, *hs*, *identifier=None*)
Bases: *qnet.algebra.operator_algebra.LocalOperator*

Unitary coherent displacement operator

$$D_{\rm s}(\alpha) := \exp\left(\alpha a_{\rm s}^\dagger - \alpha^* a_{\rm s}\right)$$

where $a_{\rm s}$ is the annihilation operator acting on the local space s.

Parameters

- **space** (LocalSpace *or* str) – Associated local Hilbert space.
- **alpha** (Any from *SCALAR_TYPES*) – Displacement amplitude.

args

all_symbols()

class qnet.algebra.operator_algebra.**Squeeze**(*eta*, ***, *hs*, *identifier=None*)
Bases: *qnet.algebra.operator_algebra.LocalOperator*

A unitary Squeezing operator acting on a particular local space/degree of freedom:

$$S_{\rm s}(\eta) := \exp\left(\frac{\eta}{2} {a_{\rm s}^\dagger}^2 - \frac{\eta^*}{2} {a_{\rm s}}^2\right)$$

where $a_{\rm s}$ is the annihilation operator acting on the local space s.

Parameters

- **space** (LocalSpace *or* str) – Associated local Hilbert space.
- **eta** (Any from *SCALAR_TYPES*) – Squeeze parameter.

args

all_symbols()

class qnet.algebra.operator_algebra.**LocalSigma**(*j*, *k*, ***, *hs*, *identifier=None*)
Bases: *qnet.algebra.operator_algebra.LocalOperator*

A local level flip operator operator acting on a particular local space/degree of freedom.

$$\sigma_{jk}^{\rm s} := \left| j\right\rangle_{\rm s} \left \langle k \right|_{\rm s}$$

Parameters

- **space** (LocalSpace *or* str) – Associated local Hilbert space.
- **j** (*int or str*) – State label j.
- **k** (*int or str*) – State label k.

args

class qnet.algebra.operator_algebra.**OperatorPlus**(**operands*, ***kwargs*)
Bases: *qnet.algebra.operator_algebra.OperatorOperation*

A sum of Operators

Parameters **operands** (*list*) – Operator summands

neutral_element = ZeroOperator

order_key
alias of FullCommutativeHSOrder

class qnet.algebra.operator_algebra.**OperatorTimes**(**operands*, ***kwargs*)
Bases: *qnet.algebra.operator_algebra.OperatorOperation*

A product of Operators that serves both as a product within a Hilbert space as well as a tensor product.

Parameters **operands** (*list*) – Operator factors

neutral_element = IdentityOperator

order_key
alias of DisjunctCommutativeHSOrder

factor_for_space(*spc*)

class qnet.algebra.operator_algebra.**ScalarTimesOperator**(**operands*, ***kwargs*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

Multiply an operator by a scalar coefficient.

Parameters

- **coefficient** (Any of *SCALAR_TYPES*) – Scalar coefficient.
- **term** (Operator) – The operator that is multiplied.

static has_minus_prefactor(*c*)
For a scalar object c, determine whether it is prepended by a "-" sign.

space

coeff

term

all_symbols()

class qnet.algebra.operator_algebra.**OperatorTrace**(*op*, ***, *over_space*)
Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

Take the (partial) trace of an operator X over the degrees of freedom given by a Hilbert hs $\mathcal{H}$:

$$\mathrm{Tr}_{\mathcal{H}} X$$

Use as:

```
OperatorTrace(X, over_space=hs)
```

Parameters

- **over_space** (HilbertSpace) – The degrees of freedom to trace over
- **op** (Operator) – The operator to take the trace of.

kwargs

operand

space

all_symbols()

class `qnet.algebra.operator_algebra.`**`Adjoint`**(*op*, ***kwargs*)

Bases: *qnet.algebra.operator_algebra.SingleOperatorOperation*

The symbolic Adjoint of an operator.

```
Adjoint(op)
```

Parameters **op** (`Operator`) – The operator to take the adjoint of.

class `qnet.algebra.operator_algebra.`**`OperatorPlusMinusCC`**(*op*, *, *sign=1*)

Bases: *qnet.algebra.operator_algebra.SingleOperatorOperation*

An operator plus or minus its complex conjugate

kwargs

minimal_kwargs

class `qnet.algebra.operator_algebra.`**`PseudoInverse`**(*op*, ***kwargs*)

Bases: *qnet.algebra.operator_algebra.SingleOperatorOperation*

The symbolic pseudo-inverse X^+ of an operator X. It is defined via the relationship

$$XX^+X = X$$
$$X^+XX^+ = X^+$$
$$(X^+X)^\dagger = X^+X$$
$$(XX^+)^\dagger = XX^+$$

Parameters **X** (`Operator`) – The operator to take the adjoint of.

class `qnet.algebra.operator_algebra.`**`NullSpaceProjector`**(*op*, ***kwargs*)

Bases: *qnet.algebra.operator_algebra.SingleOperatorOperation*

Returns a projection operator $\mathcal{P}_{\mathrm{Ker} X}$ that projects onto the nullspace of its operand

$$X\mathcal{P}_{\mathrm{Ker} X} = 0 \Leftrightarrow X(1 - \mathcal{P}_{\mathrm{Ker} X}) = X$$
$$\mathcal{P}_{\mathrm{Ker} X}^\dagger = \mathcal{P}_{\mathrm{Ker} X} = \mathcal{P}_{\mathrm{Ker} X}^2$$

Parameters **X** (`Operator`) – Operator argument

`qnet.algebra.operator_algebra.`**`LocalProjector`**(*state*, *, *hs*)

`qnet.algebra.operator_algebra.`**`X`**(*local_space*, *states=('h', 'g')*)

Pauli-type X-operator

Parameters

- **local_space** (`LocalSpace`) – Associated Hilbert space.
- **states** (*tuple with two elements of type int or str*) – The qubit state labels for the basis states $\{|0\rangle, |1\rangle\}$, where $Z|0\rangle = +|0\rangle$, default = `('h', 'g')`.

Returns Local X-operator.

Return type *Operator*

`qnet.algebra.operator_algebra.`**`Y`**(*local_space*, *states=('h', 'g')*)

Pauli-type Y-operator

Parameters

- **local_space** (`LocalSpace`) – Associated Hilbert space.

- **states** (*tuple with two elements of type int or str*) – The qubit state labels for the basis states $\{|0\rangle, |1\rangle\}$, where $Z|0\rangle = +|0\rangle$, default = `('h', 'g')`.

Returns Local Y-operator.

Return type *Operator*

`qnet.algebra.operator_algebra.`**`Z`**(*local_space*, *states=('h', 'g')*)

Pauli-type Z-operator

Parameters

- **local_space** (`LocalSpace`) – Associated Hilbert space.
- **states** (*tuple with two elements of type int or str*) – The qubit state labels for the basis states $\{|0\rangle, |1\rangle\}$, where $Z|0\rangle = +|0\rangle$, default = `('h', 'g')`.

Returns Local Z-operator.

Return type *Operator*

`qnet.algebra.operator_algebra.`**`factor_for_trace`**(*ls*, *op*)

Given a local space ls to take the partial trace over and an operator, factor the trace such that operators acting on disjoint degrees of freedom are pulled out of the trace. If the operator acts trivially on ls the trace yields only a pre-factor equal to the dimension of ls. If there are LocalSigma operators among a product, the trace's cyclical property is used to move to sandwich the full product by *LocalSigma* operators:

$$\mathrm{Tr} A \sigma_{jk} B = \mathrm{Tr} \sigma_{jk} B A \sigma_{jj}$$

Parameters

- **ls** (`HilbertSpace`) – Degree of Freedom to trace over
- **op** (`Operator`) – Operator to take the trace of

Returns The (partial) trace over the operator's spc-degrees of freedom

Return type *Operator*

`qnet.algebra.operator_algebra.`**`decompose_space`**(*H*, *A*)

Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated partial traces.

Parameters **H** (`ProductSpace`) – The full space.

Returns Iterative partial trace expression

Return type *Operator*

`qnet.algebra.operator_algebra.`**`get_coeffs`**(*expr*, *expand=False*, *epsilon=0.0*)

Create a dictionary with all Operator terms of the expression (understood as a sum) as keys and their coefficients as values.

The returned object is a defaultdict that return 0. if a term/key doesn't exist. :param expr: The operator expression to get all coefficients from. :param expand: Whether to expand the expression distributively. :param epsilon: If non-zero, drop all Operators with coefficients that have absolute value less than epsilon. :return: A dictionary of {op1: coeff1, op2: coeff2, ...} :rtype: dict

`qnet.algebra.operator_algebra.`**`space`**(*obj*)

Gives the associated HilbertSpace with an object. Also works for *SCALAR_TYPES*

`qnet.algebra.operator_algebra.`**`simplify_scalar`**(*s*)

Simplify all occurences of scalar expressions in s

Parameters **s** (`Expression` *or SympyBasic*) – The expression to simplify.

Returns The simplified version.

Return type *Expression* or SympyBasic

`qnet.algebra.operator_algebra.`**`scalar_free_symbols`**`(*operands)`
Return all free symbols from any symbolic operand

`qnet.algebra.operator_algebra.`**`factor_coeff`**`(cls, ops, kwargs)`
Factor out coefficients of all factors.

`qnet.algebra.operator_algebra.`**`adjoint`**`(obj)`
Return the adjoint of an obj.

`qnet.algebra.operator_algebra.`**`Jpjmcoeff`**`(ls, m)`

`qnet.algebra.operator_algebra.`**`Jzjmcoeff`**`(ls, m)`

`qnet.algebra.operator_algebra.`**`Jmjmcoeff`**`(ls, m)`

`qnet.algebra.operator_algebra.`**`create_operator_pm_cc`**`()`
Return a list of rules that can be used in an `extra_binary_rules()` context for `OperatorPlus` in order to combine suitable terms into a `OperatorPlusMinusCC` instance:

```
>>> A = OperatorSymbol('A', hs=1)
>>> sum = A + A.dag()
>>> from qnet.algebra.abstract_algebra import extra_binary_rules
>>> with extra_binary_rules(OperatorPlus, create_operator_pm_cc()):
...     sum2 = sum.simplify()
>>> print(ascii(sum2))
A^(1) + c.c.
```

The inverse is done through `expand_operator_pm_cc()`:

```
>>> print(ascii(sum2.simplify(rules=expand_operator_pm_cc())))
A^(1) + A^(1)H
```

`qnet.algebra.operator_algebra.`**`expand_operator_pm_cc`**`()`
Return a list of rules that can be used in *simplify* to expand instances of OperatorPlusMinusCC

qnet.algebra.ordering module

The *ordering* package implements the default canonical ordering for sums and products of operators, states, and superoperators.

To the extent that commutativity rules allow this, the ordering defined here groups objects of the same Hilbert space together, and orders these groups in the same order that the Hilbert spaces occur in a *ProductSpace* (lexicographically/by *order_index*/by complexity). Objects within the same Hilbert space (again, assuming they commute) are ordered by the *KeyTuple* value that *expr_order_key* returns for each object. Note that *expr_order_key* defers to the object's *_order_key* property, if available. This property should be defined for all QNET Expressions, generally ordering objects according to their type, then their label (if any), then their pre-factor then any other properties.

We assume that quantum operations have either full commutativity (sums, or products of states), or commutativity of objects only in different Hilbert spaces (e.g. products of operators). The former is handled by *FullCommutativeHSOrder*, the latter by *DisjunctCommutativeHSOrder*. Theses classes serve as the *order_key* for sums and products (e.g. *OperatorPlus* and similar classes)

A user may implement a custom ordering by subclassing (or replacing) *FullCommutativeHSOrder* and/or *DisjunctCommutativeHSOrder*, and assigning their replacements to all the desired algebraic classes.

Summary

Classes:

`DisjunctCommutativeHSOrder`	Auxiliary class that generates the correct pseudo-order relation for operator products.
`FullCommutativeHSOrder`	Auxiliary class that generates the correct pseudo-order relation for operator products.
`KeyTuple`	A tuple that allows for ordering, facilitating the default ordering of Operations.

Functions:

`expr_order_key`	A default order key for arbitrary expressions

Reference

class `qnet.algebra.ordering.`**`KeyTuple`**

Bases: `tuple`

A tuple that allows for ordering, facilitating the default ordering of Operations. It differs from a normal tuple in that it falls back to string comparison if any elements are not directly comparable

`qnet.algebra.ordering.`**`expr_order_key`**(*expr*)

A default order key for arbitrary expressions

class `qnet.algebra.ordering.`**`DisjunctCommutativeHSOrder`**(*op*, *space_order=None*, *op_order=None*)

Bases: `object`

Auxiliary class that generates the correct pseudo-order relation for operator products. Only operators acting on disjoint Hilbert spaces are commuted to reflect the order the local factors have in the total Hilbert space. I.e., `sorted(factors, key=DisjunctCommutativeHSOrder)` achieves this ordering.

class `qnet.algebra.ordering.`**`FullCommutativeHSOrder`**(*op*, *space_order=None*, *op_order=None*)

Bases: `object`

Auxiliary class that generates the correct pseudo-order relation for operator products. Only operators acting on disjoint Hilbert spaces are commuted to reflect the order the local factors have in the total Hilbert space. I.e., `sorted(factors, key=FullCommutativeHSOrder)` achieves this ordering.

qnet.algebra.pattern_matching module

Patterns may be constructed by either instantiating a `Pattern` instance directly, or (preferred) by calling the `pattern()`, `pattern_head()`, or `wc()` helper routines.

The pattern may then be matched against an expression using `match_pattern()`. The result of a match is a `MatchDict` object, which evaluates to True or False in a boolean context to indicate the success or failure of the match (or alternatively, through the *success* attribute). The `MatchDict` object also maps any wildcard names to the expression that the corresponding wildcard Pattern matches.

Summary

Classes:

MatchDict	Dictionary of wildcard names to expressions.
Pattern	Pattern for matching an expression
ProtoExpr	Object representing an un-instantiated Expression that may matched by a

Functions:

match_pattern	Recursively match *expr* with the given *expr_or_pattern*, which is either a direct expression (equal to *expr* for a successful match), or an instance of *Pattern*.
pattern	'Flat' constructor for the Pattern class, where positional and keyword
pattern_head	Helper function to create a Pattern object specfically matching a *ProtoExpr*.
wc	Helper function to create a Pattern object with an emphasis on wildcard

`__all__`: *MatchDict*, *Pattern*, *match_pattern*, *pattern*, *pattern_head*, *wc*

Reference

class `qnet.algebra.pattern_matching.`**`MatchDict`**(**args*)

Bases: `collections.OrderedDict`

Dictionary of wildcard names to expressions. Once the value for a key is set, attempting to set it again with a different value raises a KeyError. The attribute *merge_lists* may be set do modify this behavior for values that are lists: If it is set to a value different from zero two lists that are set via the same key are merged. If *merge_lists* is negative, the new values are appended to the existing values; if it is positive, the new values are prepended

In a boolean context, a MatchDict always evaluates as True (even if empty, unlike a normal dictionary), unless the *success* attribute is explicitly set to False (which a failed *Pattern* matching should do)

Attributes

- **success** (*bool*) – Value of the MatchDict object in a boolean context: `bool(match) == match.success`
- **reason** (*str*) – If *success* is False, string explaining why the match failed
- **merge_lists** (*int*) – Code that indicates how to combine multiple values that are lists

`update`(**others*)

Update dict with entries from *other*. If *other* has an attribute `success=False` and `reason`, those attributes are copied as well

class `qnet.algebra.pattern_matching.`**`Pattern`**(*head=None*, *args=None*, *kwargs=None*, ***, *mode=1*, *wc_name=None*, *conditions=None*)

Bases: `object`

Pattern for matching an expression

Parameters

- **head** (*type or None*) – The type (or tuple of types) of the expression that can be matched. If None, any type of Expression matches
- **args** (*list or None*) – List or tuple of positional arguments of the matched Expression (cf. *Expression.args*). Each element is an expression (to be matched exactly) or another Pattern instance (matched recursively). If None, no arguments are checked
- **kwargs** (*dict or None*) – Dictionary of keyword arguments of the expression (cf. *Expression.kwargs*). As for *args*, each value is an expression or Pattern instance.
- **mode** (*int*) – If the pattern is used to match the arguments of an expression, code to indicate how many arguments the Pattern can consume: *Pattern.single*, *Pattern.one_or_more*, *Pattern.zero_or_more*
- **wc_name** (*str or None*) – If pattern matches an expression, key in the resulting MatchDict for the expression. If None, the match will not be recorded in the result
- **conditions** (*list of callables, or None*) – If not None, a list of callables that take *expr* and return a boolean value. If the return value os False, the pattern is determined not to match *expr*.

Note: For (sub-)patterns that occur nested in the *args* attribute of another pattern, only the first or last subpattern may have a *mode* other than *Pattern.single*. This also implies that only one of the *args* may have a *mode* other than *Pattern.single*. This restrictions ensures that patterns can be matched without backtracking, thus guaranteeing numerical efficiency.

Example

Consider the following nested circuit expression:

```
>>> from qnet.algebra.circuit_algebra import *
>>> C1 = CircuitSymbol('C1', 3)
>>> C2 = CircuitSymbol('C2', 3)
>>> C3 = CircuitSymbol('C3', 3)
>>> C4 = CircuitSymbol('C4', 3)
>>> perm1 = CPermutation((2, 1, 0))
>>> perm2 = CPermutation((0, 2, 1))
>>> concat_expr = Concatenation(
...                 (C1 << C2 << perm1),
...                 (C3 << C4 << perm2))
```

We may match this with the following pattern:

```
>>> conditions = [lambda c: c.cdim == 3,
...               lambda c: c.name[0] == 'C']
>>> A__Circuit = wc("A__", head=CircuitSymbol,
...                 conditions=conditions)
>>> C__Circuit = wc("C__", head=CircuitSymbol,
...                 conditions=conditions)
>>> B_CPermutation = wc("B", head=CPermutation)
>>> D_CPermutation = wc("D", head=CPermutation)
>>> pattern_concat = pattern(
...         Concatenation,
...         pattern(SeriesProduct, A__Circuit, B_CPermutation),
...         pattern(SeriesProduct, C__Circuit, D_CPermutation))
>>> m = pattern_concat.match(concat_expr)
```

The match returns the following dictionary:

```
>>> result = {'A': [C1, C2], 'B': perm1, 'C': [C3, C4], 'D': perm2}
>>> assert m == result
```

single = 1

one_or_more = 2

zero_or_more = 3

extended_arg_patterns()
Return an iterator over patterns for positional arguments to be matched. This yields the elements of `args`, extended by their *mode* value

match(*expr*) → qnet.algebra.pattern_matching.MatchDict
Match the given expression (recursively) and return a *MatchDict* instance that maps any wildcard names to the expressions that the corresponding wildcard pattern matches. For (sub-)pattern that have a *mode* attribute other than *Pattern.single*, the wildcard name is mapped to a list of all matched expression.

If the match is successful, the resulting *MatchDict* instance will evalute to True in a boolean context. If the match is not successful, it will evaluate as False, and the reason for failure is stored in the *reason* attribute of the *MatchDict* object.

findall(*expr*)
Return a list of all matching (sub-)expressions in *expr*

qnet.algebra.pattern_matching.**pattern**(*head*, **args*, *mode=1*, *wc_name=None*, *conditions=None*, ***kwargs*) → qnet.algebra.pattern_matching.Pattern
'Flat' constructor for the Pattern class, where positional and keyword arguments are mapped into *args* and *kwargs*, respectively. Useful for defining rules that match an instantiated Expression with specific arguments

qnet.algebra.pattern_matching.**pattern_head**(**args*, *conditions=None*, *wc_name=None*, ***kwargs*) → qnet.algebra.pattern_matching.Pattern
Helper function to create a Pattern object specfically matching a *ProtoExpr*. The patterns used associated with *_rules* and *_binary_rules* of an *Expression* subclass must be instantiated through this routine. The function does not allow to set a wildcard name (*wc_name* must not be given / be None)

qnet.algebra.pattern_matching.**wc**(*name_mode='_'*, *head=None*, *args=None*, *kwargs=None*, ***, *conditions=None*) → qnet.algebra.pattern_matching.Pattern
Helper function to create a Pattern object with an emphasis on wildcard patterns if we don't care about the arguments of the matched expressions (otherwise, use `pattern()`)

Parameters

- **name_mode** (`str`) – Combined wc_name and mode for *Pattern* constructor argument. See below for syntax
- **head** (`type, or None`) – See *Pattern*
- **args** (`list or None`) – See *Pattern*
- **kwargs** (`dict or None`) – See *Pattern*
- **conditions** (`list or None`) – See *Pattern*

The *name_mode* argument uses trailing underscored to indicate the *mode*:

- A -> `Pattern(wc_name="A", mode=Pattern.single, ...)`
- A_ -> `Pattern(wc_name="A", mode=Pattern.single, ...)`

- B__ -> `Pattern(wc_name="B", mode=Pattern.one_or_more, ...)`
- B___ -> `Pattern(wc_name="C", mode=Pattern.zero_or_more, ...)`

class `qnet.algebra.pattern_matching.`**`ProtoExpr`**(*args*, *kwargs*)

Bases: `object`

Object representing an un-instantiated Expression that may matched by a *Pattern* created via `pattern_head()`

Parameters

- **`args`** (`list`) – positional arguments that would be used in the instantiation of the Expression
- **`kwargs`** (`dict`) – keyword arguments

`qnet.algebra.pattern_matching.`**`match_pattern`**(*expr_or_pattern: object*, *expr: object*) → qnet.algebra.pattern_matching.MatchDict

Recursively match *expr* with the given *expr_or_pattern*, which is either a direct expression (equal to *expr* for a successful match), or an instance of *Pattern*.

qnet.algebra.permutations module

Summary

Exceptions:

`BadPermutationError`	Can be raised to signal that a permutation does not pass the :py:func:check_permutation test.

Functions:

`block_perm_and_perms_within_blocks`	Decompose a permutation into a block permutation and into permutations acting within each block.
`check_permutation`	Verify that a tuple of permutation image points `(sigma(1), sigma(2), ..., sigma(n))` is a valid permutation, i.e.
`compose_permutations`	Find the composite permutation
`concatenate_permutations`	Concatenate two permutations:
`full_block_perm`	Extend a permutation of blocks to a permutation for the internal signals of all blocks.
`invert_permutation`	Compute the image tuple of the inverse permutation.
`permutation_from_block_permutations`	Reverse operation to `permutation_to_block_permutations()`
`permutation_from_disjoint_cycles`	Reconstruct a permutation image tuple from a list of disjoint cycles
`permutation_to_block_permutations`	If possible, decompose a permutation into a sequence of permutations each acting on individual ranges of the full range of indices.
`permutation_to_disjoint_cycles`	Any permutation sigma can be represented as a product of cycles.
`permute`	Apply a permutation sigma({j}) to an arbitrary sequence.

Reference

exception qnet.algebra.permutations.**BadPermutationError**
Bases: ValueError

Can be raised to signal that a permutation does not pass the :py:func:check_permutation test.

qnet.algebra.permutations.**check_permutation**(*permutation*)
Verify that a tuple of permutation image points `(sigma(1), sigma(2), ..., sigma(n))` is a valid permutation, i.e. each number from 0 and n-1 occurs exactly once. I.e. the following **set**-equality must hold:

```
{sigma(1), sigma(2), ..., sigma(n)} == {0, 1, 2, ... n-1}
```

Parameters **permutation** (*tuple*) – Tuple of permutation image points

Return type bool

qnet.algebra.permutations.**invert_permutation**(*permutation*)
Compute the image tuple of the inverse permutation.

Parameters **permutation** – A valid (cf. :py:func:check_permutation) permutation.

Returns The inverse permutation tuple

Return type tuple

qnet.algebra.permutations.**permutation_to_disjoint_cycles**(*permutation*)
Any permutation sigma can be represented as a product of cycles. A cycle (c_1, .. c_n) is a closed sequence of indices such that

sigma(c_1) == c_2, sigma(c_2) == sigma^2(c_1)== c_3, ..., sigma(c_(n-1)) == c_n, sigma(c_n) == c_1

Any single length-n cycle admits n equivalent representations in correspondence with which element one defines as c_1.

(0,1,2) == (1,2,0) == (2,0,1)

A decomposition into *disjoint* cycles can be made unique, by requiring that the cycles are sorted by their smallest element, which is also the left-most element of each cycle. Note that permutations generated by disjoint cycles commute. E.g.,

(1, 0, 3, 2) == ((1,0),(3,2)) –> ((0,1),(2,3)) normal form

Parameters **permutation** (*tuple*) – A valid permutation image tuple

Returns A list of disjoint cycles, that when comb

Return type list

Raise BadPermutationError

qnet.algebra.permutations.**permutation_from_disjoint_cycles**(*cycles*, *offset=0*)
Reconstruct a permutation image tuple from a list of disjoint cycles :param cycles: sequence of disjoint cycles :type cycles: list or tuple :param offset: Offset to subtract from the resulting permutation image points :type offset: int :return: permutation image tuple :rtype: tuple

qnet.algebra.permutations.**permutation_to_block_permutations**(*permutation*)
If possible, decompose a permutation into a sequence of permutations each acting on individual ranges of the full range of indices. E.g.

```
(1,2,0,3,5,4) --> (1,2,0) [+] (0,2,1)
```

Parameters **permutation** (*tuple*) – A valid permutation image tuple `s = (s_0,...s_n)` with `n > 0`

Returns A list of permutation tuples `[t = (t_0,...,t_n1), u = (u_0,...,u_n2),..., z = (z_0,...,z_nm)]` such that `s = t [+] u [+] ... [+] z`

Return type list of tuples

Raise ValueError

`qnet.algebra.permutations.`**`permutation_from_block_permutations`**(*permutations*)

Reverse operation to `permutation_to_block_permutations()` Compute the concatenation of permutations

```
(1,2,0) [+] (0,2,1) --> (1,2,0,3,5,4)
```

Parameters **permutations** (*list of tuples*) – A list of permutation tuples `[t = (t_0,...,t_n1), u = (u_0,...,u_n2),..., z = (z_0,...,z_nm)]`

Returns permutation image tuple `s = t [+] u [+] ... [+] z`

Return type tuple

`qnet.algebra.permutations.`**`compose_permutations`**(*alpha*, *beta*)

Find the composite permutation

$$\sigma := \alpha \cdot \beta$$
$$\Leftrightarrow \sigma(j) = \alpha\left(\beta(j)\right)$$

Parameters

- **a** – first permutation image tuple
- **beta** (*tuple*) – second permutation image tuple

Returns permutation image tuple of the composition.

Return type tuple

`qnet.algebra.permutations.`**`concatenate_permutations`**(*a*, *b*)

Concatenate two permutations: s = a [+] b

Parameters

- **a** (*tuple*) – first permutation image tuple
- **b** (*tuple*) – second permutation image tuple

Returns permutation image tuple of the concatenation.

Return type tuple

`qnet.algebra.permutations.`**`permute`**(*sequence*, *permutation*)

Apply a permutation sigma({j}) to an arbitrary sequence.

Parameters

- **sequence** – Any finite length sequence `[l_1,l_2,...l_n]`. If it is a list, tuple or str, the return type will be the same.
- **permutation** (*tuple*) – permutation image tuple

Returns The permuted sequence `[l_sigma(1), l_sigma(2), ..., l_sigma(n)]`

Raise BadPermutationError or ValueError

`qnet.algebra.permutations.`**`full_block_perm`**(*block_permutation*, *block_structure*)

Extend a permutation of blocks to a permutation for the internal signals of all blocks. E.g., say we have two blocks of sizes ('block structure') `(2, 3)`, then a block permutation that switches the blocks would be given by the image tuple `(1,0)`. However, to get a permutation of all 2+3 = 5 channels that realizes that block permutation we would need `(2, 3, 4, 0, 1)`

Parameters

- **block_permutation** (*tuple*) – permutation image tuple of block indices
- **block_structure** (*tuple*) – The block channel dimensions, block structure

Returns A single permutation for all channels of all blocks.

Return type tuple

`qnet.algebra.permutations.`**`block_perm_and_perms_within_blocks`**(*permutation*, *block_structure*)

Decompose a permutation into a block permutation and into permutations acting within each block.

Parameters

- **permutation** (*tuple*) – The overall permutation to be factored.
- **block_structure** (*tuple*) – The channel dimensions of the blocks

Returns `(block_permutation, permutations_within_blocks)` Where `block_permutations` is an image tuple for a permutation of the block indices and `permutations_within_blocks` is a list of image tuples for the permutations of the channels within each block

Return type tuple

qnet.algebra.singleton module

Constant algebraic objects are best implemented as singletons (i.e., they only exist as a single object). This module provides the means to declare singletons:

- The `Singleton` metaclass ensures that every class based on it produces the same object every time it is instantiated
- The `singleton_object()` class decorator returns a singleton class definition with the actual singleton object

Singletons in QNET should use both of these.

Summary

Classes:

`Singleton`	Metaclass for singletons.

Functions:

`singleton_object`	Class decorator that transforms (and replaces) a class definition (which must have a Singleton metaclass) with the actual singleton object.

`__all__`: *Singleton*, *singleton_object*

Reference

`qnet.algebra.singleton.`**singleton_object**(*cls*)
Class decorator that transforms (and replaces) a class definition (which must have a Singleton metaclass) with the actual singleton object. Ensures that the resulting object can still be "instantiated" (i.e., called), returning the same object. Also ensures the object can be pickled, is hashable, and has the correct string representation (the name of the singleton)

class `qnet.algebra.singleton.`**Singleton**
Bases: `abc.ABCMeta`

Metaclass for singletons. Any instantiation of a Singleton class yields the exact same object, e.g.:

```
>>> class MyClass(metaclass=Singleton):
...     pass
>>> a = MyClass()
>>> b = MyClass()
>>> a is b
True
```

qnet.algebra.state_algebra module

This module implements a basic Hilbert space state algebra.

Summary

Exceptions:

OverlappingSpaces	
SpaceTooLargeError	
UnequalSpaces	

Classes:

BasisKet	Local basis state, labeled by an integer or a string.
Bra	The associated dual/adjoint state for any `Ket` object `k` is given by `Bra(k)`.
BraKet	The symbolic inner product between two states, represented as Bra and
CoherentStateKet	Local coherent state, labeled by a scalar amplitude.
Ket	Basic Ket algebra class to represent Hilbert Space states
KetBra	A symbolic operator formed by the outer product of two states
KetPlus	A sum of Ket states.
KetSymbol	Ket symbol class, parametrized by an identifier string and an associated Hilbert space.
LocalKet	A state that lives on a single local Hilbert space.
OperatorTimesKet	Multiply an operator by an operator
ScalarTimesKet	Multiply a Ket by a scalar coefficient.

Continued on next page

Table 10.24 – continued from previous page

TensorKet	A tensor product of kets each belonging to different degrees of freedom.

Functions:

act_locally	
act_locally_times_tensor	
check_kets_same_space	Check that all operands are from the same Hilbert space.
check_op_ket_space	Check that all operands are from the same Hilbert space.
tensor_decompose_kets	

__all__: *BasisKet*, *Bra*, *BraKet*, *CoherentStateKet*, *Ket*, *KetBra*, *KetPlus*, *KetSymbol*, *LocalKet*, *OperatorTimesKet*, *OverlappingSpaces*, *ScalarTimesKet*, *SpaceTooLargeError*, *TensorKet*, *TrivialKet*, *UnequalSpaces*, *ZeroKet*

Module data:

qnet.algebra.state_algebra.**TrivialKet**

qnet.algebra.state_algebra.**ZeroKet**

Reference

exception qnet.algebra.state_algebra.**UnequalSpaces**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

exception qnet.algebra.state_algebra.**OverlappingSpaces**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

exception qnet.algebra.state_algebra.**SpaceTooLargeError**
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

qnet.algebra.state_algebra.**check_kets_same_space**(*cls*, *ops*, *kwargs*)
Check that all operands are from the same Hilbert space.

qnet.algebra.state_algebra.**check_op_ket_space**(*cls*, *ops*, *kwargs*)
Check that all operands are from the same Hilbert space.

class qnet.algebra.state_algebra.**Ket**
Bases: object

Basic Ket algebra class to represent Hilbert Space states

space
The associated HilbertSpace

adjoint()
The adjoint of a Ket state, i.e., the corresponding Bra.

dag
The adjoint of a Ket state, i.e., the corresponding Bra.

expand()
Expand out distributively all products of sums. Note that this does not expand out sums of scalar coefficients.

Returns A fully expanded sum of states.

Return type *Ket*

class qnet.algebra.state_algebra.**KetSymbol**(*label*, *, *hs*)
Bases: *qnet.algebra.state_algebra.Ket*, *qnet.algebra.abstract_algebra.Expression*

Ket symbol class, parametrized by an identifier string and an associated Hilbert space.

Parameters

- **label** (*str*) – Symbol identifier
- **hs** (HilbertSpace) – Associated Hilbert space.

args

kwargs

space

label

all_symbols()

class qnet.algebra.state_algebra.**LocalKet**(*label*, *, *hs*)
Bases: *qnet.algebra.state_algebra.KetSymbol*

A state that lives on a single local Hilbert space. This does not include operations, even if these operations only involve states acting on the same local space

all_symbols()

class qnet.algebra.state_algebra.**BasisKet**(*label*, *, *hs*)
Bases: *qnet.algebra.state_algebra.LocalKet*

Local basis state, labeled by an integer or a string. Basis kets are orthornormal

Parameters

- **hs** (LocalSpace) – The local Hilbert space degree of freedom.
- **or int) label** (*(str*) – The basis state label.

class qnet.algebra.state_algebra.**CoherentStateKet**(*ampl*, *, *hs*)
Bases: *qnet.algebra.state_algebra.LocalKet*

Local coherent state, labeled by a scalar amplitude.

Parameters

- **hs** (LocalSpace) – The local Hilbert space degree of freedom.
- **amp** (*SCALAR_TYPES*) – The coherent displacement amplitude.

ampl

args

kwargs

all_symbols()

class qnet.algebra.state_algebra.**KetPlus**(**operands*)
Bases: *qnet.algebra.state_algebra.Ket*, *qnet.algebra.abstract_algebra.Operation*

A sum of Ket states.

Instantiate as:

```
KetPlus(*summands)
```

Parameters **summands** (Ket) – State summands.

neutral_element = ZeroKet

order_key
alias of FullCommutativeHSOrder

space

class qnet.algebra.state_algebra.**TensorKet**(*operands*)
Bases: *qnet.algebra.state_algebra.Ket*, *qnet.algebra.abstract_algebra.Operation*

A tensor product of kets each belonging to different degrees of freedom. Instantiate as:

```
TensorKet(*factors)
```

Parameters **factors** (Ket) – Ket factors.

neutral_element = TrivialKet

order_key
alias of FullCommutativeHSOrder

classmethod create(*ops*)

factor_for_space(*space*)
Factor into a Ket defined on the given *space* and a Ket on the remaining Hilbert space

space

label
Combined label of the product state if the state is a simple product of LocalKets, raise AttributeError otherwise

class qnet.algebra.state_algebra.**ScalarTimesKet**(*operands*, ***kwargs*)
Bases: *qnet.algebra.state_algebra.Ket*, *qnet.algebra.abstract_algebra.Operation*

Multiply a Ket by a scalar coefficient.

Instantiate as:: ScalarTimesKet(coefficient, term)

Parameters

- **coefficient** (*SCALAR_TYPES*) – Scalar coefficient.
- **term** (Ket) – The ket that is multiplied.

space

coeff

term

class qnet.algebra.state_algebra.**OperatorTimesKet**(*operands*, ***kwargs*)
Bases: *qnet.algebra.state_algebra.Ket*, *qnet.algebra.abstract_algebra.Operation*

Multiply an operator by an operator

Instantiate as:

```
OperatorTimesKet(op, ket)
```

Parameters

- **op** (Operator) – The multiplying operator.
- **ket** (Ket) – The ket that is multiplied.

space

operator

ket

class qnet.algebra.state_algebra.**Bra**(*ket*)

Bases: *qnet.algebra.abstract_algebra.Operation*

The associated dual/adjoint state for any Ket object k is given by Bra(k).

Parameters k (Ket) – The state to represent as Bra.

ket

The state that is represented as a Bra.

Return type *Ket*

operand

The state that is represented as a Bra.

Return type *Ket*

adjoint()

The adjoint of a Bra is just the original Ket again.

Return type *Ket*

dag

The adjoint of a Bra is just the original Ket again.

Return type *Ket*

space

label

class qnet.algebra.state_algebra.**BraKet**(*bra, ket*)

Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

The symbolic inner product between two states, represented as Bra and Ket

In math notation this corresponds to:

$$\langle b|k\rangle$$

which we define to be linear in the state k and anti-linear in b.

Parameters

- **bra** (Ket) – The anti-linear state argument.
- **ket** (Ket) – The linear state argument.

ket

bra

space

class qnet.algebra.state_algebra.**KetBra**(*ket*, *bra*)

Bases: *qnet.algebra.operator_algebra.Operator*, *qnet.algebra.abstract_algebra.Operation*

A symbolic operator formed by the outer product of two states

Parameters

- **ket** (Ket) – The first state that defines the range of the operator.
- **bra** (Ket) – The second state that defines the Kernel of the operator.

ket

bra

space

qnet.algebra.state_algebra.**act_locally**(*op*, *ket*)

qnet.algebra.state_algebra.**act_locally_times_tensor**(*op*, *ket*)

qnet.algebra.state_algebra.**tensor_decompose_kets**(*a*, *b*, *operation*)

qnet.algebra.super_operator_algebra module

The specification of a quantum mechanics symbolic super-operator algebra. See *The Super-Operator Algebra module* for more details.

Summary

Exceptions:

BadLiouvillianError	Raise when a Liouvillian is not of standard Lindblad form.
CannotSymbolicallyDiagonalize	

Classes:

SPost	Linear post-multiplication operator.
SPre	Linear pre-multiplication operator.
ScalarTimesSuperOperator	Multiply an operator by a scalar coefficient:
SuperAdjoint	The symbolic SuperAdjoint of a super-operator.
SuperCommutativeHSOrder	Ordering class that acts like DisjunctCommutativeHSOrder, but also
SuperOperator	The super-operator abstract base class.
SuperOperatorOperation	Base class for Operations acting only on SuperOperator arguments.
SuperOperatorPlus	A sum of super-operators.
SuperOperatorSymbol	Super-operator symbol class, parametrized by an identifier string and an associated Hilbert space.
SuperOperatorTimes	A product of super-operators that denotes order of application of

Continued on next page

Table 10.27 – continued from previous page

SuperOperatorTimesOperator	Application of a super-operator to an operator (result is an Operator).

Functions:

anti_commutator	If `B != None`, return the anti-commutator $\{A,B\}$, otherwise return the super-operator $\{A,\cdot\}$.
commutator	If `B != None`, return the commutator $[A,B]$, otherwise return the super-operator $[A,\cdot]$.
lindblad	Return `SPre(C) * SPost(C.adjoint()) - (1/2) * santi_commutator(C.adjoint()*C)`.
liouvillian	Return the Liouvillian super-operator associated with a Hamilton operator `H` and a set of collapse-operators `Ls = [L1, L2, ...]`.
liouvillian_normal_form	Return a Hamilton operator `H` and a minimal list of collapse operators `Ls` that generate the liouvillian `L`.

`__all__`: *BadLiouvillianError*, *CannotSymbolicallyDiagonalize*, *IdentitySuperOperator*, *SPost*, *SPre*, *ScalarTimesSuperOperator*, *SuperAdjoint*, *SuperCommutativeHSOrder*, *SuperOperator*, *SuperOperatorOperation*, *SuperOperatorPlus*, *SuperOperatorSymbol*, *SuperOperatorTimes*, *SuperOperatorTimesOperator*, *ZeroSuperOperator*, *anti_commutator*, *commutator*, *lindblad*, *liouvillian*, *liouvillian_normal_form*

Module data:

`qnet.algebra.super_operator_algebra.`**`IdentitySuperOperator`**

`qnet.algebra.super_operator_algebra.`**`ZeroSuperOperator`**

Reference

exception `qnet.algebra.super_operator_algebra.`**`CannotSymbolicallyDiagonalize`**

Bases: *qnet.algebra.abstract_algebra.AlgebraException*

exception `qnet.algebra.super_operator_algebra.`**`BadLiouvillianError`**

Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Raise when a Liouvillian is not of standard Lindblad form.

class `qnet.algebra.super_operator_algebra.`**`SuperOperator`**

Bases: `object`

The super-operator abstract base class.

Any super-operator contains an associated HilbertSpace object, on which it is taken to act non-trivially.

`space`

The Hilbert space associated with the operator on which it acts non-trivially

`superadjoint``()`

The super-operator adjoint (w.r.t to the `Tr` operation). See *SuperAdjoint* documentation.

Returns The super-adjoint of the super-operator.

Return type *SuperOperator*

expand()
Expand out distributively all products of sums. Note that this does not expand out sums of scalar coefficients.

Returns A fully expanded sum of superoperators.

Return type *SuperOperator*

simplify_scalar()
Simplify all scalar coefficients within the Operator expression.

Returns The simplified expression.

Return type *Operator*

class qnet.algebra.super_operator_algebra.**SuperOperatorOperation**(**operands*)
Bases: *qnet.algebra.super_operator_algebra.SuperOperator*, *qnet.algebra.abstract_algebra.Operation*

Base class for Operations acting only on SuperOperator arguments.

space

class qnet.algebra.super_operator_algebra.**SuperOperatorSymbol**(*label*, ***, *hs*)
Bases: *qnet.algebra.super_operator_algebra.SuperOperator*, *qnet.algebra.abstract_algebra.Expression*

Super-operator symbol class, parametrized by an identifier string and an associated Hilbert space.

Instantiate as:

```
SuperOperatorSymbol(name, hs)
```

Parameters

- **label** (*str*) – Symbol identifier
- **hs** (HilbertSpace) – Associated Hilbert space.

label

args

kwargs

space

all_symbols()

class qnet.algebra.super_operator_algebra.**SuperOperatorPlus**(**operands*)
Bases: *qnet.algebra.super_operator_algebra.SuperOperatorOperation*

A sum of super-operators.

Instantiate as:

```
OperatorPlus(*summands)
```

Parameters **summands** (SuperOperator) – super-operator summands.

neutral_element = ZeroSuperOperator

order_key
alias of FullCommutativeHSOrder

class qnet.algebra.super_operator_algebra.**SuperCommutativeHSOrder**(*op*, *space_order=None*, *op_order=None*)

Bases: *qnet.algebra.ordering.DisjunctCommutativeHSOrder*

Ordering class that acts like DisjunctCommutativeHSOrder, but also commutes any *SPost* and *SPre*

class qnet.algebra.super_operator_algebra.**SuperOperatorTimes**(**operands*)

Bases: *qnet.algebra.super_operator_algebra.SuperOperatorOperation*

A product of super-operators that denotes order of application of super-operators (right to left):

```
SuperOperatorTimes(*factors)
```

Parameters **factors** (SuperOperator) – Super-operator factors.

neutral_element = IdentitySuperOperator

order_key

alias of *SuperCommutativeHSOrder*

classmethod create(**ops*)

class qnet.algebra.super_operator_algebra.**ScalarTimesSuperOperator**(**operands*, ***kwargs*)

Bases: *qnet.algebra.super_operator_algebra.SuperOperator*, *qnet.algebra.abstract_algebra.Operation*

Multiply an operator by a scalar coefficient:

```
ScalarTimesSuperOperator(coeff, term)
```

Parameters

- **coeff** (*SCALAR_TYPES*) – Scalar coefficient.
- **term** (SuperOperator) – The super-operator that is multiplied.

space

coeff

The scalar coefficient.

term

The super-operator term.

class qnet.algebra.super_operator_algebra.**SuperAdjoint**(*operand*)

Bases: *qnet.algebra.super_operator_algebra.SuperOperatorOperation*

The symbolic SuperAdjoint of a super-operator.

The math notation for this is typically

$$\text{SuperAdjoint}(\mathcal{L}) =: \mathcal{L}^*$$

and for any super operator $\mathcal{L}$, its super-adjoint $\mathcal{L}^*$ satisfies for any pair of operators M, N:

$$\text{Tr}[M(\mathcal{L}N)] = Tr[(\mathcal{L}^*M)N]$$

Parameters **L** (SuperOperator) – The super-operator to take the adjoint of.

operand

class `qnet.algebra.super_operator_algebra.`**`SPre`**(*op*)
Bases: *`qnet.algebra.super_operator_algebra.SuperOperator`*, *`qnet.algebra.abstract_algebra.Operation`*

Linear pre-multiplication operator.

Acting `SPre(A)` on an operator `B` just yields the product `A * B`

`space`

class `qnet.algebra.super_operator_algebra.`**`SPost`**(*op*)
Bases: *`qnet.algebra.super_operator_algebra.SuperOperator`*, *`qnet.algebra.abstract_algebra.Operation`*

Linear post-multiplication operator.

Acting `SPost(A)` on an operator `B` just yields the reversed product `B * A`.

`space`

class `qnet.algebra.super_operator_algebra.`**`SuperOperatorTimesOperator`**(*sop*, *op*)
Bases: *`qnet.algebra.operator_algebra.Operator`*, *`qnet.algebra.abstract_algebra.Operation`*

Application of a super-operator to an operator (result is an Operator).

Parameters

- **`sop`** (`SuperOperator`) – The super-operator to apply.
- **`op`** (`Operator`) – The operator it is applied to.

`space`

`sop`

`op`

`qnet.algebra.super_operator_algebra.`**`commutator`**(*A*, *B=None*)
If `B != None`, return the commutator $[A, B]$, otherwise return the super-operator $[A, \cdot]$. The super-operator $[A, \cdot]$ maps any other operator `B` to the commutator $[A, B] = AB - BA$.

Parameters

- **`A`** (`Operator`) – The first operator to form the commutator of.
- **`or None) B`** (*(*`Operator`) – The second operator to form the commutator of, or None.

Returns The linear superoperator $[A, \cdot]$

Return type *SuperOperator*

`qnet.algebra.super_operator_algebra.`**`anti_commutator`**(*A*, *B=None*)
If `B != None`, return the anti-commutator $\{A, B\}$, otherwise return the super-operator $\{A, \cdot\}$. The super-operator $\{A, \cdot\}$ maps any other operator `B` to the anti-commutator $\{A, B\} = AB + BA$.

Parameters

- **`A`** (`Operator`) – The first operator to form all anti-commutators of.
- **`or None) B`** (*(*`Operator`) – The second operator to form the anti-commutator of, or None.

Returns The linear superoperator $[A, \cdot]$

Return type *SuperOperator*

`qnet.algebra.super_operator_algebra.`**`lindblad`**`(C)`

Return `SPre(C) * SPost(C.adjoint()) - (1/2) * santi_commutator(C.adjoint()*C)`. These are the super-operators $\mathcal{D}[C]$ that form the collapse terms of a Master-Equation. Applied to an operator X they yield

$$\mathcal{D}[C]X = CXC^\dagger - \frac{1}{2}(C^\dagger CX + XC^\dagger C)$$

Parameters **C** (`Operator`) – The associated collapse operator

Returns The Lindblad collapse generator.

Return type *SuperOperator*

`qnet.algebra.super_operator_algebra.`**`liouvillian`**`(H, Ls=[])`

Return the Liouvillian super-operator associated with a Hamilton operator `H` and a set of collapse-operators `Ls = [L1, L2, ...]`.

The Liouvillian $\mathcal{L}$ generates the Markovian-dynamics of a system via the Master equation:

$$\dot{\rho} = \mathcal{L}\rho = -i[H, \rho] + \sum_{j=1}^{n} \mathcal{D}[L_j]\rho$$

Parameters

- **H** (`Operator`) – The associated Hamilton operator
- **Ls** (*sequence or* `Matrix`) – A sequence of collapse operators.

Returns The Liouvillian super-operator.

Return type *SuperOperator*

`qnet.algebra.super_operator_algebra.`**`liouvillian_normal_form`**`(L, symbolic=False)`

Return a Hamilton operator `H` and a minimal list of collapse operators `Ls` that generate the liouvillian `L`.

A Liouvillian defined by a hermitian Hamilton operator H and a vector of collapse operators $\mathbf{L} = (L_1, L_2, \dots L_n)^T$ is invariant under the following two operations:

$$(H, \mathbf{L}) \mapsto \left(H + \frac{1}{2i}\left(\mathbf{w}^\dagger \mathbf{L} - \mathbf{L}^\dagger \mathbf{w}\right), \mathbf{L} + \mathbf{w}\right)$$

$$(H, \mathbf{L}) \mapsto (H, \mathbf{U}\mathbf{L})$$

where $\mathbf{w}$ is just a vector of complex numbers and $\mathbf{U}$ is a complex unitary matrix. It turns out that for quantum optical circuit models the set of collapse operators is often linearly dependent. This routine tries to find a representation of the Liouvillian in terms of a Hamilton operator `H` with as few non-zero collapse operators `Ls` as possible. Consider the following example, which results from a two-port linear cavity with a coherent input into the first port:

```
>>> kappa_1, kappa_2 = symbols('kappa_1, kappa_2', positive = True)
>>> Delta = symbols('Delta', real = True)
>>> alpha = symbols('alpha')
>>> H = (Delta * Create(hs=1) * Destroy(hs=1) +
...      (sqrt(kappa_1) / (2 * I)) *
...      (alpha * Create(hs=1) - alpha.conjugate() * Destroy(hs=1)))
>>> Ls = [sqrt(kappa_1) * Destroy(hs=1) + alpha,
...       sqrt(kappa_2) * Destroy(hs=1)]
>>> LL = liouvillian(H, Ls)
>>> Hnf, Lsnf = liouvillian_normal_form(LL)
>>> print(ascii(Hnf))
-I*alpha*sqrt(kappa_1) * a^(1)H + I*sqrt(kappa_1)*conjugate(alpha) * a^(1) +
↪Delta * a^(1)H * a^(1)
```

```
>>> len(Lsnf)
1
>>> print(ascii(Lsnf[0]))
sqrt(kappa_1 + kappa_2) * a^(1)
```

In terms of the ensemble dynamics this final system is equivalent. Note that this function will only work for proper Liouvillians.

Parameters `L` (SuperOperator) – The Liouvillian

Returns `(H, Ls)`

Return type tuple

Raises BadLiouvillianError

`__all__`: *ABCD*, *Adjoint*, *AlgebraError*, *AlgebraException*, *BadLiouvillianError*, *BasisKet*, *BasisNotSetError*, *Bra*, *BraKet*, CIdentity, *CPermutation*, *CannotConvertToABCD*, *CannotConvertToSLH*, *CannotEliminateAutomatically*, *CannotSimplify*, *CannotSymbolicallyDiagonalize*, *CannotVisualize*, *Circuit*, *CircuitSymbol*, CircuitZero, *CoherentStateKet*, *Concatenation*, *Create*, *Destroy*, *Displace*, *Expression*, *FB*, *Feedback*, FullSpace, *HilbertSpace*, II, IdentityOperator, IdentitySuperOperator, *ImAdjoint*, *ImMatrix*, *IncompatibleBlockStructures*, *Jminus*, *Jmjmcoeff*, *Jpjmcoeff*, *Jplus*, *Jz*, *Jzjmcoeff*, *Ket*, *KetBra*, *KetPlus*, *KetSymbol*, *LocalKet*, *LocalOperator*, *LocalProjector*, *LocalSigma*, *LocalSpace*, *MatchDict*, *Matrix*, *NonSquareMatrix*, *NullSpaceProjector*, *Operation*, *Operator*, *OperatorOperation*, *OperatorPlus*, *OperatorPlusMinusCC*, *OperatorSymbol*, *OperatorTimes*, *OperatorTimesKet*, *OperatorTrace*, *OverlappingSpaces*, *P_sigma*, *Pattern*, *Phase*, *ProductSpace*, *PseudoInverse*, *ReAdjoint*, *ReMatrix*, SCALAR_TYPES, *SLH*, *SPost*, *SPre*, *ScalarTimesKet*, *ScalarTimesOperator*, *ScalarTimesSuperOperator*, *SeriesInverse*, *SeriesProduct*, *SingleOperatorOperation*, *SpaceTooLargeError*, *Squeeze*, *SuperAdjoint*, *SuperCommutativeHSOrder*, *SuperOperator*, *SuperOperatorOperation*, *SuperOperatorPlus*, *SuperOperatorSymbol*, *SuperOperatorTimes*, *SuperOperatorTimesOperator*, *TensorKet*, TrivialKet, TrivialSpace, *UnequalSpaces*, *WrongCDimError*, *WrongSignatureError*, *X*, *Y*, *Z*, ZeroKet, ZeroOperator, ZeroSuperOperator, *adjoint*, *all_symbols*, *anti_commutator*, *block_matrix*, *cid*, cid_1, *circuit_identity*, *commutator*, *connect*, *create_operator_pm_cc*, *decompose_space*, *diagm*, *eval_adiabatic_limit*, *expand_operator_pm_cc*, *extra_binary_rules*, *extra_rules*, *extract_signal*, *extract_signal_circuit*, *factor_coeff*, *factor_for_trace*, *getABCD*, *get_coeffs*, *get_common_block_structure*, *hstackm*, *identity_matrix*, *lindblad*, *liouvillian*, *liouvillian_normal_form*, *map_signals*, *map_signals_circuit*, *match_pattern*, *move_drive_to_H*, *no_instance_caching*, *no_rules*, *pad_with_identity*, *pattern*, *pattern_head*, *permutation_matrix*, *prepare_adiabatic_limit*, *scalar_free_symbols*, *set_union*, *simplify*, *simplify_scalar*, *space*, *substitute*, *temporary_instance_cache*, *try_adiabatic_elimination*, *vstackm*, *wc*, *zerosm*

qnet.circuit_components package

This module contains all defined *primitive* circuit component definitions as well as the compiled circuit definitions that are automatically created via the `$QNET/bin/parse_qhdl.py` script. For some examples on how to create your own circuit definition file, check out the source code to

- *qnet.circuit_components.single_sided_jaynes_cummings_cc*
- *qnet.circuit_components.three_port_opo_cc*
- *qnet.circuit_components.kerr_cavity_cc*

The module *qnet.circuit_components.component* features some base classes for component definitions and the module *qnet.circuit_components.library* features some utility functions to help manage the circuit component definitions.

Submodules:

qnet.circuit_components.and_cc module

And component

Summary

Classes:

And

`__all__`: *And*

Reference

class `qnet.circuit_components.and_cc.`**`And`**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 4

Delta = 50.0

chi = -0.26

kappa_1 = 20.0

kappa_2 = 20.0

kappa_3 = 10.0

theta = 0.6435

phi = -1.39

phip = 2.65

PORTSIN = ['In1', 'In2']

PORTSOUT = ['Out1']

B1

B2

C

Phase1

Phase2

space

qnet.circuit_components.beamsplitter_cc module

Component definition file for a infinite bandwidth beamsplitter with variable mixing angle. See *Beamsplitter*

Summary

Classes:

Beamsplitter	Infinite bandwidth beamsplitter model.

`__all__`: *Beamsplitter*

Reference

class `qnet.circuit_components.beamsplitter_cc.`**`Beamsplitter`**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

Infinite bandwidth beamsplitter model. It is a pure scattering component, i.e. it's internal dynamics are not modeled explicitly. The single real parameter theta is the mixing angle for the two signals. Note that there is an asymmetry in the effect on the two input signals due to the minus sign appearing in the scattering matrix

$$S = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

To achieve a more general beamsplitter combine this component with one or more `qnet.circuit_components.Phase` components.

Instantiate as:

```
Beamsplitter("B", theta = pi/4)
```

`CDIM` **= 2**

`theta` **= pi/4**

`PORTSIN` **= ['In1', 'In2']**

`PORTSOUT` **= ['Out1', 'Out2']**

qnet.circuit_components.component module

We distinguish between two independent properties of Components:

1) They may be 'creducible', i.e. they can be expressed as a circuit expression of sub components.

2. They may be 'primitive', i.e. they cannot be specified via QHDL

We write 'creducible' instead of 'reducible' in order to distinguish the meaning from the definition of Gough and James, who define reducible circuits as all circuits that can be decomposed into a concatenation of parts. Creducibility is more general than reducibility since we allow for an expansion into any sort of non-trivial algebraic expression, but in reverse, all reducible circuits are creducible.

Examples of creducible but primitive Components are: KerrCavity, Relay, ...

non-creducible & primitive: Beamsplitter, Phase, Displace

creducible & non-primitive: Any parsed QHDL circuit

non-creducible & non-primitive: None.

Summary

Classes:

Component	Base class for all circuit components, both primitive components such as beamsplitters and cavity models and also composite circuit models that are built up from these.
SubComponent	Class for the subcomponents of a reducible (but primitive) Component.

`__all__`: *Component*, *SubComponent*

Reference

class `qnet.circuit_components.component.`**`Component`**(*name*, ***kwargs*)

Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Expression*

Base class for all circuit components, both primitive components such as beamsplitters and cavity models and also composite circuit models that are built up from these. Via the creduce() method, an object can be decomposed into its parts.

CDIM = 0

PORTSIN = []

PORTSOUT = []

name

args

kwargs

cdim

space

class `qnet.circuit_components.component.`**`SubComponent`**(*parent_component*, *sub_index*)

Bases: *qnet.algebra.circuit_algebra.Circuit*, *qnet.algebra.abstract_algebra.Expression*

Class for the subcomponents of a reducible (but primitive) Component.

parent_component = None

sub_index = 0

PORTSIN

Names of ingoing ports.

PORTSOUT

Names of outgoing ports.

cdim

Numbers of channels

name

args

all_symbols()

space

qnet.circuit_components.delay_cc module

Component definition file for a pseudo-delay model that works over a limited bandwidth. See documentation of *Delay*.

Summary

Classes:

Delay	

__all__: *Delay*

Reference

class qnet.circuit_components.delay_cc.**Delay**(*name*, ***kwargs*)
Bases: *qnet.circuit_components.component.Component*

CDIM = 1

tau = tau

N = 3

FOCK_DIM = 25

PORTSIN = ['In1']

PORTSOUT = ['Out1']

qnet.circuit_components.displace_cc module

Component definition file for a coherent field displacement component. See documentation of *Displace*.

Summary

Classes:

Displace	Coherent displacement of the input field (usually vacuum) by a complex amplitude α.

__all__: *Displace*

Reference

class `qnet.circuit_components.displace_cc.`**`Displace`**(*name*, ***kwargs*)

Bases: `qnet.circuit_components.component.Component`

Coherent displacement of the input field (usually vacuum) by a complex amplitude α. This component serves as the model of an ideal laser source without internal non-classical internal dynamics.

`CDIM` **= 1**

`alpha` **= alpha**

`PORTSIN` **= ['VacIn']**

`PORTSOUT` **= ['Out1']**

qnet.circuit_components.double_sided_jaynes_cummings_cc module

Component definition file for a two mirror CQED Jaynes-Cummings cavity model.

See documentation of `DoubleSidedJaynesCummings`.

Summary

Classes:

`CavityPort`	Sub component model for port coupling the internal mode of a `DoubleSidedJaynesCummings` model to the external field.
`DecayChannel`	Sub component model for the port coupling the internal two-level atom to the vacuum of the transverse free-field modes, inducing spontaneous emission/decay.
`DoubleSidedJaynesCummings`	Typical CQED Jaynes-Cummings model with a two laser input/output channels with coupling coefficients κ_1 and κ_2, respectively, and a single atomic decay channel with rate γ.

`__all__`: `DoubleSidedJaynesCummings`

Reference

class `qnet.circuit_components.double_sided_jaynes_cummings_cc.`**`DoubleSidedJaynesCummings`**(*name*, ***kwa*

Bases: `qnet.circuit_components.component.Component`

Typical CQED Jaynes-Cummings model with a two laser input/output channels with coupling coefficients κ_1 and κ_2, respectively, and a single atomic decay channel with rate γ. The full model is given by:

$$S = \mathbf{1}_3$$

$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_1} a \\ \sqrt{\gamma} \sigma_- \end{pmatrix}$$

$$H = \Delta_f a^\dagger a + \Delta_a \sigma_+ \sigma_- + ig \left(\sigma_+ a - \sigma_- a^\dagger \right)$$

As the model is reducible, sub component models for the mode and the atomic decay channel are given by `CavityPort` and `DecayChannel`, respectively.

CDIM = 3

kappa_1 = kappa_1

kappa_2 = kappa_2

gamma = gamma

g = g

Delta_a = Delta_a

Delta_f = Delta_f

FOCK_DIM = 20

PORTSIN = ['In1', 'In2', 'VacIn']

PORTSOUT = ['Out1', 'Out2', 'UOut']

sub_blockstructure = (1, 1, 1)

fock_space
The cavity mode's Hilbert space.

Type *qnet.algebra.hilbert_space_algebra.LocalSpace*

tls_space
The two-level-atom's Hilbert space.

Type *qnet.algebra.hilbert_space_algebra.LocalSpace*

space

class qnet.circuit_components.double_sided_jaynes_cummings_cc.**CavityPort**(*parent_component*, *sub_index*)
Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for port coupling the internal mode of a *DoubleSidedJaynesCummings* model to the external field. The Hamiltonian is included with this first port.

class qnet.circuit_components.double_sided_jaynes_cummings_cc.**DecayChannel**(*cavity*)
Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for the port coupling the internal two-level atom to the vacuum of the transverse free-field modes, inducing spontaneous emission/decay.

qnet.circuit_components.double_sided_opo_cc module

Component definition file for a degenerate OPO model with two signal beam ports. See documentation of *DoubleSidedOPO*.

Summary

Classes:

DoubleSidedOPO	This model describes a degenerate OPO with two signal beam ports in the sub-threshold regime.
OPOPort	Sub component model for the individual ports of a *DoubleSidedOPO*.

__all__: *DoubleSidedOPO*

Reference

class qnet.circuit_components.double_sided_opo_cc.**DoubleSidedOPO**(*name*, ***kwargs*)

Bases: `qnet.circuit_components.component.Component`

This model describes a degenerate OPO with two signal beam ports in the sub-threshold regime. I.e., the pump is modeled as a classical amplitude.

The model's SLH parameters are given by

$$S = \mathbf{1}_2$$

$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} a \end{pmatrix}$$

$$H = \Delta a^\dagger a + \frac{i}{2}\left(\alpha {a^\dagger}^2 - \alpha^* a^2\right)$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate `OPOPort` models for each port.

Note that this model's validity breaks down even in open-loop configuration when

$$|\alpha| > \frac{\kappa_1 + \kappa_2}{2}$$

which is just the threshold condition. In a feedback configuration the threshold condition is generally changed.

CDIM = 2

kappa_1 = kappa_1

kappa_2 = kappa_2

alpha = alpha

Delta = Delta

FOCK_DIM = 25

PORTSIN = ['In1', 'In2']

PORTSOUT = ['Out1', 'Out2']

sub_blockstructure = (1, 1)

space

class qnet.circuit_components.double_sided_opo_cc.**OPOPort**(*parent_component*, *sub_index*)

Bases: `qnet.circuit_components.component.SubComponent`

Sub component model for the individual ports of a `DoubleSidedOPO`. The Hamiltonian is included with the first port.

qnet.circuit_components.inverting_fanout_cc module

Summary

Classes:

`InvertingFanout`

__all__: *InvertingFanout*

Reference

class qnet.circuit_components.inverting_fanout_cc.**InvertingFanout**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 5

Delta = 50.0

chi = -0.14

kappa_1 = 20.0

kappa_2 = 20.0

kappa_3 = 10.0

theta = 0.473

phi = -1.45

phip = -0.49

alpha = -130.0

PORTSIN = ['In1']

PORTSOUT = ['Out1', 'Out2']

B1

B2

B3

C

Phase1

Phase2

W

space

qnet.circuit_components.kerr_cavity_cc module

Component definition file for a Kerr-nonlinear cavity model with two ports. See documentation of *KerrCavity*.

Summary

Classes:

KerrCavity	This model describes a Kerr cavity model with two ports.
KerrPort	Sub component model for the individual ports of a *KerrCavity*.

__all__: *KerrCavity*

Reference

class qnet.circuit_components.kerr_cavity_cc.**KerrCavity**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This model describes a Kerr cavity model with two ports.

The model's SLH parameters are given by

$$S = \mathbf{1}_2$$
$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} a \end{pmatrix}$$
$$H = \Delta a^\dagger a + \chi {a^\dagger}^2 a^2$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate *KerrPort* models for each port.

CDIM = 2

PORTSIN = ['In1', 'In2']

PORTSOUT = ['Out1', 'Out2']

sub_blockstructure = (1, 1)

Delta = Delta

chi = chi

kappa_1 = kappa_1

kappa_2 = kappa_2

FOCK_DIM = 75

space

port1

port2

class qnet.circuit_components.kerr_cavity_cc.**KerrPort**(*parent_component*, *sub_index*)

Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for the individual ports of a *KerrCavity*. The Hamiltonian is included with the first port.

qnet.circuit_components.latch_cc module

Summary

Classes:

Latch

__all__: *Latch*

Reference

class `qnet.circuit_components.latch_cc.`**`Latch`**(*name*, ***kwargs*)

Bases: *`qnet.circuit_components.component.Component`*

`CDIM` **= 8**

`Delta` **= 50.0**

`chi` **= -0.205**

`kappa_1` **= 20.0**

`kappa_2` **= 20.0**

`kappa_3` **= 10.0**

`theta` **= 0.891**

`thetap` **= 0.593**

`phi` **= 2.72**

`phip` **= 0.14**

`beta` **= (-79.838356622-35.806239846j)**

`PORTSIN` **= ['In1', 'In2']**

`PORTSOUT` **= ['Out1']**

`B11`

`B12`

`B21`

`B22`

`B3`

`C1`

`C2`

`Phase1`

`Phase2`

`Phase3`

`W1`

`W2`

`space`

qnet.circuit_components.library module

This module features some helper functions for automatically creating and managing a library of circuit component definition files.

Summary

Functions:

`camelcase_to_underscore`	Convert a camelcase entity name into an appropriate underscore name to import its corresponding module
`getCDIM`	Get the channel dimension of a referenced subcomponent
`make_namespace_string`	Make a namespace string by combining a namespace string with a new name.
`write_component`	Write a new entity definition to a python module file.

Reference

qnet.circuit_components.library.**make_namespace_string**(*namespace*, *sub_name*)
Make a namespace string by combining a namespace string with a new name.

Parameters

- **namespace** (*str*) – The namespace so far
- **sub_name** (*str*) – The additional name to add/

Returns The combined namespace

Return type str

qnet.circuit_components.library.**camelcase_to_underscore**(*st*)
Convert a camelcase entity name into an appropriate underscore name to import its corresponding module

qnet.circuit_components.library.**getCDIM**(*component_name*)
Get the channel dimension of a referenced subcomponent

Parameters **component_name** (*str*) – The entity name of the component

Returns The channel dimension of the component.

Return type int

qnet.circuit_components.library.**write_component**(*entity*, *architectures*, *local=False*)
Write a new entity definition to a python module file.

Parameters

- **entity** (*qnet.qhdl.qhdl.Entity*) – The entity object
- **architectures** (*dict*) – A dictionary of architectures `dict(name = architecture)` associated with the entity.
- **local** (*bool*) – Whether or not to store the created module in the current/local directory or install it in :py:module:`qnet.circuit_components`, default = `False`

Returns The filename of the new module.

Return type str

qnet.circuit_components.linear_cavity_cc module

Component definition file for a simple linear cavity model with two ports. See documentation of `LinearCavity`.

Summary

Classes:

CavityPort	Sub component model for the individual ports of a LinearCavity.
LinearCavity	This model describes a cavity model with two ports.

__all__: LinearCavity

Reference

class qnet.circuit_components.linear_cavity_cc.**LinearCavity**(*name*, ***kwargs*)

Bases: qnet.circuit_components.component.Component

This model describes a cavity model with two ports.

The model's SLH parameters are given by

$$S = \mathbf{1}_2$$
$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} a \end{pmatrix}$$
$$H = \Delta a^\dagger a$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate CavityPort models for each port.

CDIM = 2

PORTSIN = ['In1', 'In2']

PORTSOUT = ['Out1', 'Out2']

sub_blockstructure = (1, 1)

Delta = Delta

kappa_1 = kappa_1

kappa_2 = kappa_2

FOCK_DIM = 75

space

port1

port2

class qnet.circuit_components.linear_cavity_cc.**CavityPort**(*parent_component*, *sub_index*)

Bases: qnet.circuit_components.component.SubComponent

Sub component model for the individual ports of a LinearCavity. The Hamiltonian is included with the first port.

qnet.circuit_components.mach_zehnder_cc module

Summary

Classes:

MachZehnder

__all__: *MachZehnder*

Reference

class qnet.circuit_components.mach_zehnder_cc.**MachZehnder**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 2

alpha = alpha

phi = phi

PORTSIN = ['a', 'b']

PORTSOUT = ['c', 'd']

B1

B2

P

W

space

qnet.circuit_components.open_lossy_cc module

Summary

Classes:

OpenLossy

__all__: *OpenLossy*

Reference

class qnet.circuit_components.open_lossy_cc.**OpenLossy**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 3

Delta = Delta

chi = chi

kappa = kappa

theta = theta

theta_LS0 = theta_LS0

PORTSIN = ['In1']

PORTSOUT = ['Out1', 'Out2']

BS

KC

LSS_ci_ls

space

qnet.circuit_components.phase_cc module

Component definition file for a coherent field Phasement component. See documentation of *Phase*.

Summary

Classes:

Phase	Coherent phase shift of the field passing through by real angle ϕ.

__all__: *Phase*

Reference

class qnet.circuit_components.phase_cc.**Phase** (*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

Coherent phase shift of the field passing through by real angle ϕ.

CDIM = 1

phi = phi

PORTSIN = ['In1']

PORTSOUT = ['Out1']

qnet.circuit_components.pseudo_nand_cc module

Summary

Classes:

PseudoNAND

__all__: *PseudoNAND*

Reference

class qnet.circuit_components.pseudo_nand_cc.**PseudoNAND** (*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 4

Delta = Delta

chi = chi

kappa = kappa

phi = phi

theta = theta

beta = beta

PORTSIN = ['A', 'B', 'VIn1', 'VIn2']

PORTSOUT = ['UOut1', 'UOut2', 'NAND_AB', 'OUT2']

BS1

BS2

K

P

W_beta

space

qnet.circuit_components.pseudo_nand_latch_cc module

Summary

Classes:

PseudoNANDLatch

`__all__`: *PseudoNANDLatch*

Reference

class qnet.circuit_components.pseudo_nand_latch_cc.**PseudoNANDLatch**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

CDIM = 6

PORTSIN = ['NS', 'W1', 'kerr2_extra', 'NR', 'W2', 'kerr1_extra']

PORTSOUT = ['BS1_1_out', 'kerr1_out2', 'OUT2_2', 'BS1_2_out', 'kerr2_out2', 'OUT2_1']

NAND1

NAND2

space

qnet.circuit_components.relay_cc module

Component definition file for an all-optical Relay model. See documentation of *Relay*.

Summary

Classes:

Relay	This is the Relay model as used in our group's QEC papers[1],[#qec2]_.
RelayControl	Second subcomponent of a *Relay* model.
RelayOut	First subcomponent of a *Relay* model.

`__all__`: *Relay*

Reference

class `qnet.circuit_components.relay_cc.`**`Relay`**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This is the Relay model as used in our group's QEC papers[1],[#qec2]_. The `SET` and `RESET` inputs control whether the `POW` input is routed through the `OUT` or the `NOUT` output port.

Since the scattering matrix is of block diagonal form (2x2,2x2) we provide sub component models for the individual subsystems *RelayOut* and *RelayControl*.

`CDIM` = 4

`PORTSIN` = ['POW', 'VIn', 'SET', 'RESET']

`PORTSOUT` = ['NOUT', 'OUT', 'UOut1', 'UOut2']

`sub_blockstructure` = (2, 2)

`space`

class `qnet.circuit_components.relay_cc.`**`RelayOut`**(*relay*)

Bases: *qnet.circuit_components.component.SubComponent*

First subcomponent of a *Relay* model.

class `qnet.circuit_components.relay_cc.`**`RelayControl`**(*relay*)

Bases: *qnet.circuit_components.component.SubComponent*

Second subcomponent of a *Relay* model.

qnet.circuit_components.relay_double_probe_cc module

Component definition file for an all-optical Relay model. See documentation of `Relay`.

Summary

Classes:

RelayControl	Second subcomponent of a `Relay` model.
RelayDoubleProbe	This is the Relay model as used in our group's QEC papers[1],[#qec2]_.
RelayOut	First subcomponent of a `Relay` model.

[1] http://pra.aps.org/abstract/PRA/v80/i4/e045802

`__all__`: *RelayDoubleProbe*

Reference

class `qnet.circuit_components.relay_double_probe_cc.`**`RelayDoubleProbe`**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This is the Relay model as used in our group's QEC papers[1],[#qec2]_. The `SET` and `RESET` inputs control whether the `POW` input is routed through the `OUT` or the `NOUT` output port.

Since the scattering matrix is of block diagonal form (2x2,2x2) we provide sub component models for the individual subsystems *RelayOut* and *RelayControl*.

`CDIM` = 6

`PORTSIN` = ['POW1', 'VIn1', 'POW2', 'VIn2', 'SET', 'RESET']

`PORTSOUT` = ['NOUT1', 'OUT1', 'NOUT2', 'OUT2', 'UOut1', 'UOut2']

`sub_blockstructure` = (2, 2, 2)

`space`

class `qnet.circuit_components.relay_double_probe_cc.`**`RelayOut`**(*parent_component*, *sub_index*)

Bases: *qnet.circuit_components.component.SubComponent*

First subcomponent of a `Relay` model.

class `qnet.circuit_components.relay_double_probe_cc.`**`RelayControl`**(*parent_component*, *sub_index*)

Bases: *qnet.circuit_components.component.SubComponent*

Second subcomponent of a `Relay` model.

qnet.circuit_components.single_sided_jaynes_cummings_cc module

Component definition file for a single mirror CQED Jaynes-Cummings cavity model.

See documentation of *SingleSidedJaynesCummings*.

Summary

Classes:

CavityPort	Sub component model for port coupling the internal mode of a *SingleSidedJaynesCummings* model to the external field.
DecayChannel	Sub component model for the port coupling the internal two-level atom to the vacuum of the transverse free-field modes, inducing spontaneous emission/decay.
SingleSidedJaynesCummings	Typical CQED Jaynes-Cummings model with a single laser input/output channel with coupling coefficient κ and a single atomic decay channel with rate γ.

[1] http://pra.aps.org/abstract/PRA/v80/i4/e045802

`__all__`: *SingleSidedJaynesCummings*

Reference

class qnet.circuit_components.single_sided_jaynes_cummings_cc.**SingleSidedJaynesCummings**(*name*, ***kwa*

Bases: *qnet.circuit_components.component.Component*

Typical CQED Jaynes-Cummings model with a single laser input/output channel with coupling coefficient κ and a single atomic decay channel with rate γ. The full model is given by:

$$S = \mathbf{1}_2$$
$$L = \begin{pmatrix} \sqrt{\kappa}a \\ \sqrt{\gamma}\sigma_- \end{pmatrix}$$
$$H = \Delta_f a^\dagger a + \Delta_a \sigma_+ \sigma_- + ig\left(\sigma_+ a - \sigma_- a^\dagger\right)$$

As the model is reducible, sub component models for the mode and the atomic decay channel are given by *CavityPort* and *DecayChannel*, respectively.

CDIM = 2

kappa = kappa

gamma = gamma

g = g

Delta_a = Delta_a

Delta_f = Delta_f

FOCK_DIM = 20

PORTSIN = ['In1', 'VacIn']

PORTSOUT = ['Out1', 'UOut']

sub_blockstructure = (1, 1)

fock_space
The cavity mode's Hilbert space.

Type *qnet.algebra.hilbert_space_algebra.LocalSpace*

tls_space
The two-level-atom's Hilbert space.

Type *qnet.algebra.hilbert_space_algebra.LocalSpace*

class qnet.circuit_components.single_sided_jaynes_cummings_cc.**CavityPort**(*cavity*)
Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for port coupling the internal mode of a *SingleSidedJaynesCummings* model to the external field. The Hamiltonian is included with this first port.

class qnet.circuit_components.single_sided_jaynes_cummings_cc.**DecayChannel**(*cavity*)
Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for the port coupling the internal two-level atom to the vacuum of the transverse free-field modes, inducing spontaneous emission/decay.

qnet.circuit_components.single_sided_opo_cc module

Component definition file for a degenerate OPO model with a single port for the signal beam. See documentation of *SingleSidedOPO*.

Summary

Classes:

SingleSidedOPO	This model describes a degenerate OPO with a single port for the signal mode in the sub-threshold regime: i.e., the pump is modeled as a classical amplitude.

`__all__`: *SingleSidedOPO*

Reference

class qnet.circuit_components.single_sided_opo_cc.**SingleSidedOPO**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This model describes a degenerate OPO with a single port for the signal mode in the sub-threshold regime: i.e., the pump is modeled as a classical amplitude.

The model's SLH parameters are given by

$$S = (1)$$
$$L = \left(\sqrt{\kappa} a\right)$$
$$H = \Delta a^\dagger a + \frac{i}{2}\left(\alpha {a^\dagger}^2 - \alpha^* a^2\right)$$

CDIM = 1

kappa = kappa

alpha = alpha

Delta = Delta

FOCK_DIM = 25

PORTSIN = ['In1']

PORTSOUT = ['Out1']

space

qnet.circuit_components.three_port_kerr_cavity_cc module

Component definition file for a Kerr-nonlinear cavity model with two ports. See documentation of *ThreePortKerrCavity*.

Summary

Classes:

`KerrPort`	Sub component model for the individual ports of a `ThreePortKerrCavity`.
`ThreePortKerrCavity`	This model describes a Kerr cavity model with three ports.

`__all__`: `ThreePortKerrCavity`

Reference

class `qnet.circuit_components.three_port_kerr_cavity_cc.`**`ThreePortKerrCavity`**(*name*, ***kwargs*)

Bases: `qnet.circuit_components.component.Component`

This model describes a Kerr cavity model with three ports.

The model's SLH parameters are given by

$$S = \mathbf{1}_3$$
$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} a \\ \sqrt{\kappa_3} a \end{pmatrix}$$
$$H = \Delta a^\dagger a + \chi {a^\dagger}^2 a^2$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate `KerrPort` models for each port.

`CDIM` = **3**

`PORTSIN` = **['In1', 'In2', 'In3']**

`PORTSOUT` = **['Out1', 'Out2', 'Out3']**

`sub_blockstructure` = **(1, 1, 1)**

`Delta` = **Delta**

`chi` = **chi**

`kappa_1` = **kappa_1**

`kappa_2` = **kappa_2**

`kappa_3` = **kappa_3**

`FOCK_DIM` = **75**

`space`

`port1`

`port2`

`port3`

class `qnet.circuit_components.three_port_kerr_cavity_cc.`**`KerrPort`**(*parent_component*, *sub_index*)

Bases: `qnet.circuit_components.component.SubComponent`

Sub component model for the individual ports of a `ThreePortKerrCavity`. The Hamiltonian is included with the first port.

qnet.circuit_components.three_port_opo_cc module

Component definition file for a degenerate OPO model with three signal beam ports. See documentation of *ThreePortOPO*.

Summary

Classes:

OPOPort	Sub component model for the individual ports of a *ThreePortOPO*.
ThreePortOPO	This model describes a degenerate OPO with three signal beam ports in the sub-threshold regime.

`__all__`: *ThreePortOPO*

Reference

class `qnet.circuit_components.three_port_opo_cc.`**`ThreePortOPO`**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This model describes a degenerate OPO with three signal beam ports in the sub-threshold regime. I.e., the pump is modeled as a classical amplitude.

The model's SLH parameters are given by

$$S =_3$$

$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} a \\ \sqrt{\kappa_3} a \end{pmatrix}$$

$$H = \Delta a^\dagger a + \frac{i}{2}\left(\alpha {a^\dagger}^2 - \alpha^* a^2\right)$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate *OPOPort* models for each port.

Note that this model's validity breaks down even in open-loop configuration when

$$|\alpha| > \frac{\kappa_1 + \kappa_2 + \kappa_3}{2}$$

which is just the threshold condition. In a feedback configuration the threshold condition is generally changed.

CDIM = 3

kappa_1 = kappa_1

kappa_2 = kappa_2

kappa_3 = kappa_2

alpha = alpha

Delta = Delta

FOCK_DIM = 25

PORTSIN = ['In1', 'In2', 'In3']

PORTSOUT = ['Out1', 'Out2', 'In3']

sub_blockstructure = (1, 1, 1)

space

class qnet.circuit_components.three_port_opo_cc.**OPOPort** (*parent_component*, *sub_index*)

Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for the individual ports of a *ThreePortOPO*. The Hamiltonian is included with the first port.

qnet.circuit_components.two_port_kerr_cavity_cc module

Component definition file for a Kerr-nonlinear cavity model with two ports. See documentation of *TwoPortKerrCavity*.

Summary

Classes:

KerrPort	Sub component model for the individual ports of a *TwoPortKerrCavity*.
TwoPortKerrCavity	This model describes a Kerr cavity model with two ports.

__all__: *TwoPortKerrCavity*

Reference

class qnet.circuit_components.two_port_kerr_cavity_cc.**TwoPortKerrCavity** (*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This model describes a Kerr cavity model with two ports.

The model's SLH parameters are given by

$$S = \mathbf{1}_2$$
$$L = \begin{pmatrix} \sqrt{\kappa_1} a \\ \sqrt{\kappa_2} \end{pmatrix}$$
$$H = \Delta a^\dagger a + \chi {a^\dagger}^2 a^2$$

This particular component definition explicitly captures the reducibility of a trivial scattering matrix. I.e., it can be reduced into separate *KerrPort* models for each port.

CDIM = 2

PORTSIN = ['In1', 'In2']

PORTSOUT = ['Out1', 'Out2']

sub_blockstructure = (1, 1, 1)

Delta = Delta

chi = chi

kappa_1 = kappa_1

kappa_2 = kappa_2

FOCK_DIM = 75

space

port1

port2

class qnet.circuit_components.two_port_kerr_cavity_cc.**KerrPort**(*parent_component*, *sub_index*)

Bases: *qnet.circuit_components.component.SubComponent*

Sub component model for the individual ports of a *TwoPortKerrCavity*. The Hamiltonian is included with the first port.

qnet.circuit_components.z_probe_cavity_cc module

Component definition file for the Z-probe cavity model from the Mabuchi-Lab Coherent Feedback Quantum Error Correction papers.

See documentation of *ZProbeCavity*.

Summary

Classes:

FeedbackPort	Feedback beam port for the Z-Probe cavity model.
LossPort	Spontaneous decay from the far detuned excited r level.
ProbePort	Probe beam port for the Z-Probe cavity model.
ZProbeCavity	This is the Z-probe cavity model as used in our group's QEC papers [#qec1,#qec2]_ , which has three (dressed) internal states: r, g, h.

__all__: *ZProbeCavity*

Reference

class qnet.circuit_components.z_probe_cavity_cc.**ZProbeCavity**(*name*, ***kwargs*)

Bases: *qnet.circuit_components.component.Component*

This is the Z-probe cavity model as used in our group's QEC papers [#qec1,#qec2]_ , which has three (dressed) internal states: r, g, h. The qubit is encoded in g,h, while r is used to drive transitions. The first channel is the probe-signal, while the second and third channels are the two independent feedback beams.

Since the scattering matrix is diagonal we provide sub component models for the individual subsystems: One *ProbePort* and two *FeedbackPort* instances..

CDIM = 5

gamma = gamma

gamma_p = gamma_p

Delta = Delta

PORTSIN = ['PIn', 'FIn1', 'FIn2']

PORTSOUT = ['POut']

sub_blockstructure = (1, 1, 1, 1, 1)

space

class qnet.circuit_components.z_probe_cavity_cc.**ProbePort**(*cavity*)
Bases: *qnet.circuit_components.component.SubComponent*

Probe beam port for the Z-Probe cavity model.

class qnet.circuit_components.z_probe_cavity_cc.**FeedbackPort**(*parent_component*, *sub_index*)
Bases: *qnet.circuit_components.component.SubComponent*

Feedback beam port for the Z-Probe cavity model.

class qnet.circuit_components.z_probe_cavity_cc.**LossPort**(*parent_component*, *sub_index*)
Bases: *qnet.circuit_components.component.SubComponent*

Spontaneous decay from the far detuned excited r level.

__all__: *And*, *Beamsplitter*, *Component*, *Delay*, *Displace*, *DoubleSidedJaynesCummings*, *DoubleSidedOPO*, *InvertingFanout*, *KerrCavity*, *Latch*, *LinearCavity*, *MachZehnder*, *OpenLossy*, *Phase*, *PseudoNAND*, *PseudoNANDLatch*, *Relay*, *RelayDoubleProbe*, *SingleSidedJaynesCummings*, *SingleSidedOPO*, *SubComponent*, *ThreePortKerrCavity*, *ThreePortOPO*, *TwoPortKerrCavity*, *ZProbeCavity*

qnet.convert package

Submodules:

qnet.convert.to_qutip module

Conversion of QNET expressions to qutip objects.

Summary

Functions:

SLH_to_qutip	Generate and return QuTiP representation matrices for the Hamiltonian and the collapse operators.
convert_to_qutip	Convert a QNET expression to a qutip object

__all__: *SLH_to_qutip*, *convert_to_qutip*

Reference

qnet.convert.to_qutip.**convert_to_qutip**(*expr*, *full_space=None*, *mapping=None*)
Convert a QNET expression to a qutip object

Parameters

- **expr** – a QNET expression
- **full_space** (HilbertSpace) – The Hilbert space in which *expr* is defined. If not given, `expr.space` is used. The Hilbert space must have a well-defined basis.
- **mapping** (*dict*) – A mapping of any (sub-)expression to either a *quip.Qobj* directly, or to a callable that will convert the expression into a *qutip.Qobj*. Useful for e.g. supplying objects for symbols

Raises ValueError – if *expr* is not in *full_space*, or if *expr* cannot be converted.

`qnet.convert.to_qutip.`**`SLH_to_qutip`**(*slh*, *full_space=None*, *time_symbol=None*, *convert_as='pyfunc'*)

Generate and return QuTiP representation matrices for the Hamiltonian and the collapse operators. Any inhomogeneities in the Lindblad operators (resulting from coherent drives) will be moved into the Hamiltonian, cf. *move_drive_to_H()*.

Parameters

- **slh** (SLH) – The SLH object from which to generate the qutip data
- **full_space** (HilbertSpace *or None*) – The Hilbert space in which to represent the operators. If None, the space of *shl* will be used
- **time_symbol** (`sympy.Symbol` or None) – The symbol (if any) expressing time dependence (usually 't')
- **convert_as** (*str*) – How to express time dependencies to qutip. Must be 'pyfunc' or 'str'

Returns tuple `(H, [L1, L2, ...])` as numerical *qutip.Qobj* representations, where `H` and each `L` may be a nested list to express time dependence, e.g. `H = [H0, [H1, eps_t]]`, where `H0` and `H1` are of type *qutip.Qobj*, and `eps_t` is either a string (`convert_as='str'`) or a function (`convert_as='pyfunc'`)

Raises `AlgebraError` – If the Hilbert space (*slh.space* or *full_space*) is invalid for numerical conversion

qnet.convert.to_sympy_matrix module

Conversion of QNET expressions to sympy matrices. For small Hilbert spaces, this facilitates some analytic treatments, such as decomposition into a basis.

Summary

Functions:

SympyCreate	Creation operator for a Hilbert space of dimension *n*, as an instance
basis_state	`n x 1` *sympy.Matrix* representing the *i*'th eigenstate of an
convert_to_sympy_matrix	Convert a QNET expression to an explicit `n x n` instance of *sympy.Matrix*, where `n` is the dimension of *full_space*.

`__all__`: *convert_to_sympy_matrix*

Reference

qnet.convert.to_sympy_matrix.**basis_state**(*i*, *n*)
`n x 1` *sympy.Matrix* representing the *i*'th eigenstate of an *n*-dimensional Hilbert space (*i* >= 0)

qnet.convert.to_sympy_matrix.**SympyCreate**(*n*)
Creation operator for a Hilbert space of dimension *n*, as an instance of *sympy.Matrix*

qnet.convert.to_sympy_matrix.**convert_to_sympy_matrix**(*expr*, *full_space=None*)
Convert a QNET expression to an explicit `n x n` instance of *sympy.Matrix*, where `n` is the dimension of *full_space*. The entries of the matrix may contain symbols.

Parameters

- **expr** – a QNET expression
- **full_space** (`qnet.algebra.hilbert_space_algebra.HilbertSpace`) – The Hilbert space in which *expr* is defined. If not given, `expr.space` is used. The Hilbert space must have a well-defined basis.

Raises

- *qnet.algebra.hilbert_space_algebra.BasisNotSetError* – if *full_space* does not have a defined basis
- `ValueError` – if *expr* is not in *full_space*, or if *expr* cannot be converted.

`__all__`: *SLH_to_qutip*, *convert_to_qutip*, *convert_to_sympy_matrix*

qnet.misc package

Submodules:

qnet.misc.circuit_visualization module

Summary

Functions:

draw_circuit	Generate a graphic representation of circuit and store them in a file.
draw_circuit_canvas	Generate a PyX graphical representation of a circuit expression object.

`__all__`: *draw_circuit*, *draw_circuit_canvas*

Reference

qnet.misc.circuit_visualization.**draw_circuit_canvas**(*circuit*, *hunit=4*, *vunit=-1.0*, *rhmargin=0.1*, *rvmargin=0.2*, *rpermutation_length=0.4*, *draw_boxes=True*, *permutation_arrows=False*)
Generate a PyX graphical representation of a circuit expression object.

Parameters

- **circuit** (*ca.Circuit*) – The circuit expression
- **hunit** (*float*) – The horizontal length unit, default = HUNIT
- **vunit** (*float*) – The vertical length unit, default = VUNIT
- **rhmargin** (*float*) – relative horizontal margin, default = RHMARGIN
- **rvmargin** (*float*) – relative vertical margin, default = RVMARGIN
- **rpermutation_length** (*float*) – the relative length of a permutation circuit, default = RPLENGTH
- **draw_boxes** (*bool*) – Whether to draw indicator boxes to denote subexpressions (Concatenation, SeriesProduct, etc.), default = True
- **permutation_arrows** (*bool*) – Whether to draw arrows within the permutation visualization, default = False

Returns A PyX canvas object that can be further manipulated or printed to an output image.

Return type pyx.canvas.canvas

qnet.misc.circuit_visualization.**draw_circuit**(*circuit*, *filename*, *direction='lr'*, *hunit=4*, *vunit=-1.0*, *rhmargin=0.1*, *rvmargin=0.2*, *rpermutation_length=0.4*, *draw_boxes=True*, *permutation_arrows=False*)

Generate a graphic representation of circuit and store them in a file. The graphics format is determined from the file extension.

Parameters

- **circuit** (*ca.Circuit*) – The circuit expression
- **filename** (*str*) – A filepath to store the output image under. The file name suffix determines the output graphics format
- **direction** – The horizontal direction of laying out series products. One of 'lr' and 'rl'. This option overrides a negative value for hunit, default = 'lr'
- **hunit** (*float*) – The horizontal length unit, default = HUNIT
- **vunit** (*float*) – The vertical length unit, default = VUNIT
- **rhmargin** (*float*) – relative horizontal margin, default = RHMARGIN
- **rvmargin** (*float*) – relative vertical margin, default = RVMARGIN
- **rpermutation_length** (*float*) – the relative length of a permutation circuit, default = RPLENGTH
- **draw_boxes** (*bool*) – Whether to draw indicator boxes to denote subexpressions (Concatenation, SeriesProduct, etc.), default = True
- **permutation_arrows** (*bool*) – Whether to draw arrows within the permutation visualization, default = False

Returns True if printing was successful, False if not.

Return type bool

qnet.misc.euler_mayurama module

Summary

Functions:

`euler_mayurama_complex`	Evaluate the Ito-SDE

Reference

`qnet.misc.euler_mayurama.`**`euler_mayurama_complex`**(*f*, *g*, *x0*, *m*, *h*, *steps_per_sample*, *n_samples*, *include_initial=False*)

Evaluate the Ito-SDE

$$dx = f(x,t)dt + g(x,t)dA_t$$

using an Euler-Mayurama scheme with fixed stepsize h.

Parameters

- **f** – function for drift term
- **g** – function for diffusion term
- **x0** – initial value
- **m** – number of complex noises
- **steps_per_sample** – The number of h-sized steps between samples
- **n_samples** – the total number of integration steps
- **include_initial** – whether or not to include the initial value (and time) in the output.

Returns a tuple `(times, xs, dAs)`, where `times` is an array of times at which x is evaluated, `xs` is x at different times, and `dAs` are the noise increments for the time interval $[t, t+h)$

qnet.misc.kerr_model_matrices module

Summary

Functions:

`T_a_qp`	Basis transfer matrix, `[[a],[a.conjugate()]] = T_a_qp.dot(qp),`
`T_qp_a`	Basis transfer matrix, `qp = T_qp_a.dot([[a],[a.conjugate()]])`
`model_matrices`	Return the matrices necessary to carry out a semi-classical simulation of the SLH system driven by some dynamic inputs.
`model_matrices_complex`	Same as `model_matrices()` but tries to convert all output to purely
`model_matrices_symbolic`	Same as `model_matrices()` but converts all output to `Matrix` objects.

Continued on next page

Table 10.58 – continued from previous page

prepare_sde	Compute the SDE functions f, g and (optionally) the Jacobian of f (see *euler_mayurama* docs) for the model matrices.
substitute_into_symbolic_model_matrices	
wrap_Jqp	Wrap the jacobian of a complex ode function `f(a,t)` as `f(qp, t)`,
wrap_fqp	Wrap a complex ode function `f(a,t)` as `f(qp, t)` where

Reference

`qnet.misc.kerr_model_matrices.`**`model_matrices`**(*slh*, *dynamic_input_ports*, *apply_kerr_diagonal_correction=True*, *epsilon=0.0*, *return_eoms=False*)

Return the matrices necessary to carry out a semi-classical simulation of the SLH system driven by some dynamic inputs.

Parameters

- **slh** – SLH object
- **dynamic_input_ports** (*dict*) – Mapping of port index to `input_name_str`
- **apply_kerr_diagonal_correction** (*bool*) – whether there should be an effective detuning of 2 chi for every kerr-cavity.
- **epsilon** (*float*) – for non-zero epsilon (and a numerical coefficient slh) remove expressions with coefficents smaller than epsilon.
- **return_eoms** (*bool*) – Whether to also return the symbolic e.o.m.'s as well as the output processes.

Returns

A tuple `(A, B, C, D, A_kerr, B_input, D_input, u_c, U_c[, eoms, dA'])`

- `A`: coupling of modes to each other
- `B`: coupling of external input fields to modes
- `C`: coupling of internal modes to output
- `D`: coupling of external input fields to output fields
- `A_kerr`: kerr-type coupling between modes
- `B_input`: coupling of dynamic inputs to modes
- `D_input`: coupling of dynamic inputs to external output fields
- `u_c`: constant coherent input driving to modes
- `U_c`: constant coherent input contribution to output field
- `eoms`: symbolic QSDEs for the internal modes (if *return_eoms* is True, None otherwise)
- `dA`: symbolic expression for the output fields (if *return_eoms* is True, None otherwise)

The overall SDE is then:

$$da_t/dt = (Aa_t + (A_{kerr}(a_t \odot a_t^*)) \odot a_t + u_c + B_{input}u_t) + BdA_t/dt \quad (10.3)$$
$$dA_t'/dt = (C * a_t + U_c + D_{input}u_t) + DdA_t/dt \quad (10.4)$$

where $\odot$ denotes the element-wise product of two vectors. It is assumed that all degrees of freedom are cavities with their only non-linearity being of the Kerr-type, i.e. either self coupling $H_{kerr} = a^* a^* aa$ or cross-coupling $H_{kerr} = a^* ab^* b$.

`qnet.misc.kerr_model_matrices.`**`model_matrices_complex`**(**args*, ***kwargs*)
Same as `model_matrices()` but tries to convert all output to purely numerical matrices

`qnet.misc.kerr_model_matrices.`**`model_matrices_symbolic`**(**args*, ***kwargs*)
Same as `model_matrices()` but converts all output to `Matrix` objects.

`qnet.misc.kerr_model_matrices.`**`substitute_into_symbolic_model_matrices`**(*model_matrices*, *params*)

`qnet.misc.kerr_model_matrices.`**`prepare_sde`**(*numeric_model_matrices*, *input_fn*, *return_jac=False*)
Compute the SDE functions f, g and (optionally) the Jacobian of f (see *euler_mayurama* docs) for the model matrices.

Returns `f, g[, Jf]`

The overall SDE is:

$$da_t/dt = (Aa_t + (A_{kerr}(a_t \odot a_t^*) \odot a_t + u_c + B_{input}u_t) + BdA_t/dt \quad (10.5)$$
$$dA_t'/dt = (Ca_t + U_c + D_{input}u_t) + DdA_t/dt \quad (10.6)$$

`qnet.misc.kerr_model_matrices.`**`wrap_fqp`**(*f*)
Wrap a complex ode function `f(a,t)` as `f(qp, t)` where `qp = [a1r, a1i, a2r, a2i,...]`

`qnet.misc.kerr_model_matrices.`**`T_qp_a`**(*n*)
Basis transfer matrix, `qp = T_qp_a.dot([[a],[a.conjugate()]])`

`qnet.misc.kerr_model_matrices.`**`T_a_qp`**(*n*)
Basis transfer matrix, `[[a],[a.conjugate()]] = T_a_qp.dot(qp)`, where `qp = [a1r, a1i, a2r, a2i,...]`

`qnet.misc.kerr_model_matrices.`**`wrap_Jqp`**(*J*)
Wrap the jacobian of a complex ode function `f(a,t)` as `f(qp, t)`, where `qp = [a1r, a1i, a2r, a2i,...]`

qnet.misc.parse_circuit_strings module

Parse strings into Circuit expressions. See documentation for `parse_circuit_strings()`

Summary

Exceptions:

`ParseCircuitStringError`	Raised when an error is encountered while parsing a circuit expression string.

Functions:

parse_circuit_strings	Parse strings for symbolic Circuit expressions into actual expression objects.

Reference

qnet.misc.parse_circuit_strings.**parse_circuit_strings**(*circuit_string*)

Parse strings for symbolic Circuit expressions into actual expression objects.

Parameters **circuit_string** – A string containing one or more circuit expressions in the special syntax described below.

Returns A list of all parsed expressions if there are more than one, otherwise just the single result.

Return type list or ca.Circuit

Examples

1. A circuit symbol can be instantiated via:

```
>>> print(srepr(parse_circuit_strings('name(3)')))
CircuitSymbol('name', 3)
```

2. A concatenation can be instantiated by using the infix '+' operator

```
>>> print(srepr(parse_circuit_strings('a(3) + b(5)')))
Concatenation(CircuitSymbol('a', 3), CircuitSymbol('b', 5))
```

3. A series product can be instantiated by using the infix '<<' operator

```
>>> print(srepr(parse_circuit_strings('a(3) << b(3)')))
SeriesProduct(CircuitSymbol('a', 3), CircuitSymbol('b', 3))
```

4. Circuit identity objects for `n` channels can be instantiated via `cid(n)`:

```
>>> print(srepr(parse_circuit_strings('(a(3) + cid(1)) << b(4)')))
SeriesProduct(Concatenation(CircuitSymbol('a', 3), CIdentity), CircuitSymbol(
↪'b', 4))
```

5. Feedback operations are specified as

```
>>> print(srepr(parse_circuit_strings('[a(5)]_(1->2)')))
Feedback(CircuitSymbol('a', 5), out_port=1, in_port=2)
```

6. Permutation objects are specified as

```
>>> print(srepr(parse_circuit_strings('P_sigma(1,2,3,0)')))
CPermutation((1, 2, 3, 0))
```

exception qnet.misc.parse_circuit_strings.**ParseCircuitStringError**

Bases: *qnet.misc.parser.ParsingError*

Raised when an error is encountered while parsing a circuit expression string.

qnet.misc.parser module

Generic Parser class.

Summary

Exceptions:

ParsingError	Raised for parsing error.

Classes:

Parser	Base class for a lexer/parser that has the _rules defined as methods

Reference

exception qnet.misc.parser.**ParsingError**
Bases: SyntaxError

Raised for parsing error.

class qnet.misc.parser.**Parser**(***kw*)
Bases: object

Base class for a lexer/parser that has the _rules defined as methods

tokens = ()

precedence = ()

parse(*inputstring*)

parse_file(*filename*)

qnet.misc.parser__CircuitExpressionParser_parsetab module

qnet.misc.qsd_codegen module

Summary

Exceptions:

QSDCodeGenError	Exception raised for missing data in a *QSDCodeGen* instance

Classes:

QSDCCodePrinter	A printer for converting SymPy expressions to C++ code, while taking
QSDCodeGen	Class that allows to generate a QSD program for QNET expressions, and

Continued on next page

Table 10.64 – continued from previous page

QSDOperator	Encapsulation of a QSD (symbolic) Operator, containing all information required to instantiate that operator and to use it in C++ code expressions.

Functions:

compilation_worker	Worker to perform compilation, suitable e.g.
expand_cmd	Return a copy of the array *cmd*, where for each element of the *cmd*
find_kets	Given a Ket instance, return the set of LocalKet instances contained in it.
local_ops	Given a symbolic expression, extract the set of "atomic" operators (instances of Operator) occurring in that expression.
qsd_run_worker	Worker to perform run of a previously compiled program (see compilation_worker()), suitable e.g.
sanitize_name	Return a sanitized *name*, where all letters that occur as keys in

Reference

class qnet.misc.qsd_codegen.**QSDCCodePrinter**(*settings={}*)

Bases: sympy.printing.ccode.CCodePrinter

A printer for converting SymPy expressions to C++ code, while taking into account pre-defined variable names for symbols

qnet.misc.qsd_codegen.**local_ops**(*expr*)

Given a symbolic expression, extract the set of "atomic" operators (instances of Operator) occurring in that expression. The set is "atomic" in the sense that the operators are not algebraic combinations of other operators.

qnet.misc.qsd_codegen.**find_kets**(*expr*, *cls=<class 'qnet.algebra.state_algebra.LocalKet'>*)

Given a Ket instance, return the set of LocalKet instances contained in it.

class qnet.misc.qsd_codegen.**QSDOperator**(*qsd_type*, *name*, *instantiator*)

Bases: object

Encapsulation of a QSD (symbolic) Operator, containing all information required to instantiate that operator and to use it in C++ code expressions.

All arguments set the corresponding properties.

Examples

```
>>> A0 = QSDOperator('AnnihilationOperator', 'A0', '(0)')
>>> Ad0 = QSDOperator('Operator', 'Ad0', '= A0.hc()')
```

known_types = ['AnnihilationOperator', 'FieldTransitionOperator', 'IdentityOperator', 'Operator']

qsd_type

QSD object type, i.e., name of the C++ class. See known_types class attribute for allowed type names

name

The name of the operator object. Must be a valid C++ variable name.

instantiator
String that instantiates the operator object. This must either be the constructor arguments of the operator's QSD class, or a C++ expression (starting with an equal sign) that initializes the object

instantiation
Complete line of C++ code that instantiates the operator

Example

```
>>> A0 = QSDOperator('AnnihilationOperator', 'A0', '(0)')
>>> print(A0.instantiation)
AnnihilationOperator A0(0);
```

__iter__()
Split *QSDOperator* into a tuple.

Example

```
>>> A0 = QSDOperator('AnnihilationOperator', 'A0', '(0)')
>>> qsd_type, name, instantiator = A0
```

__str__()
The string representation of an operator is simply its name

Example

```
>>> A0 = QSDOperator('AnnihilationOperator', 'A0', '(0)')
>>> assert(str(A0) == str(A0.name))
```

exception qnet.misc.qsd_codegen.**QSDCodeGenError**
Bases: Exception

Exception raised for missing data in a *QSDCodeGen* instance

class qnet.misc.qsd_codegen.**QSDCodeGen**(*circuit*, *num_vals=None*, *time_symbol=None*)
Bases: object

Class that allows to generate a QSD program for QNET expressions, and to run the program to (accumulative) collect expectation values for observables

Parameters

- **circuit** (*SLH*) – The circuit to be simulated via QSD.
- **num_vals** (dict of Symbol to float)) – Numeric value for any symbol occurring in the *circuit*, or any operator/state that may be added later on.
- **time_symbol** (None or Symbol) – symbol to denote the time dependence in the Hamiltonian (usually *t*). If None, the Hamiltonian is time-independent.

Attributes

- **circuit** (*SLH*) – see *circuit* parameter
- **time_symbol** (None or Symbol) – see *time_symbol* parameter

- **syms** (set of `Symbol`) – The set of symbols used either in the circuit, any of the observables, or the initial state, excluding *time_symbol*
- **num_vals** (dict of `Symbol` to float)) – Map of symbols to numeric value. Must specify a value for any symbol in *syms*.
- **traj_data** (`TrajectoryData`) – The accumulated trajectory data. Every time the `run()`, respectively the `run_delayed()` method is called, the resulting trajectory data is incorporated. Thus, by repeatedly calling `run()` (followed by `run_delayed()` if `delay=True`), an arbitrary number of trajectories may be accumulated in *traj_data*.

known_steppers = ['Order4Step', 'AdaptiveStep', 'AdaptiveJump', 'AdaptiveOrthoJump']

observables
Iterator over all defined observables (instances of `Operator`)

observable_names
Iterator of all defined observable names (str)

compile_cmd
Command to be used for compilation (after `compile()` method has been called). Environment variables and '~' are not expanded

get_observable(*name*)
Return the observable for the given name (instance of `Operator`), according to the mapping defined by `add_observable()`

add_observable(*op*, *name=None*)
Register an operator as an observable, together with a name that will be used in the header of the table of expectation values, and on which the name of the QSD output files will be based.

Parameters

- **op** (`Operator`) – Observable (does not need to be Hermitian)
- **name** (str or `None`) – Name of of the operator, to be used in the header of the output table. If `None`, `str(op)` is used.

Raises `ValueError` – if *name* is invalid or too long, or no unique filename can be generated from *name*

set_moving_basis(*move_dofs*, *delta=0.0001*, *width=2*, *move_eps=0.0001*)
Activate the use of the moving basis, see Section 6 of the QSD Paper.

Parameters

- **move_dofs** (`int`) – degrees of freedom for which to use a moving basis (the first 'move_dofs' freedoms are re-centered, and their cutoffs adjusted.)
- **delta** (`float`) – probability threshold for the cutoff adjustment
- **width** (`int`) – size of the "pad" for the cutoff
- **move_eps** (`float`) – numerical accuracy with which to make the shift. Cf. `shiftAccuracy` in QSD `State::recenter` method

Raises

- `ValueError` – if *move_dofs* is invalid
- `QSDCodeGenError` – if requesting a moving basis for a degree of freedom for which any operator is defined that cannot be applied in the moving basis

set_trajectories(*psi_initial*, *stepper*, *dt*, *nt_plot_step*, *n_plot_steps*, *n_trajectories*, *traj_save=10*)
Set the parameters that control the trajectories from which a plot of expectation values for the registered observables will be generated.

Parameters

- **psi_initial** (*Ket*) – The initial state
- **stepper** (*str*) – Name of the QSD stepper that should handle propagation of a single time step. See *known_steppers* for allowed values
- **dt** (*float*) – The duration for a single propagation step. Note that the plot of expectation values will generally be on a coarser grid, as controlled by the `set_plotting` routine
- **nt_plot_step** (*int*) – Number of propagation steps per plot step. That is, expectation values of the observables will be written out every *nt_plot_step* propagation steps
- **n_plot_steps** (*int*) – Number of plot steps. The total number of propagation steps for each trajectory will be `nt_plot_step * n_plot_steps`, and duration T of the entire trajectory will be `dt * nt_plot_step * n_plot_steps`
- **n_trajectories** (*int*) – The number of trajectories over which to average for getting the expectation values of the observables
- **traj_save** (*int*) – Number of trajectories to propagate before writing the averaged expectation values of all observables to file. This ensures that if the program is terminated before the calculation of `n_trajectories` is complete, the lost data is at most that of the last `traj_save` trajectories is lost. A value of 0 indicates that the values are to be written out only after completing all trajectories.

generate_code()
Return C++ program that corresponds to the circuit as a multiline string

write(*outfile*)
Write C++ program that corresponds to the circuit

compile(*qsd_lib*, *qsd_headers*, *executable='qsd_run'*, *path='.'*, *compiler='g++'*, *compile_options='-O2'*, *delay=False*, *keep_cc=False*, *remote_apply=None*)
Compile into an executable

Parameters

- **qsd_lib** (*str*) – full path to the file `libqsd.a` containing the statically compiled QSD library. May reference environment variables the home directory ('~')
- **qsd_headers** (*str*) – path to the folder containing the QSD header files. May reference environment variables the home directory ('~')
- **executable** (*str*) – name of executable to which the QSD program should be compiled. Must consist only of letters, numbers, dashes, and underscores only
- **path** (*str*) – The path to the folder where executable will be generated. May reference environment variables the home directory ('~')
- **compiler** (*str*) – compiler executable
- **compile_options** (*str*) – options to pass to the compiler
- **delay** (*bool*) – Deprecated, must be False
- **keep_cc** (*bool*) – If True, keep the C++ code from which the executable was compiled. It will have the same name as the executable, with an added '.cc' file extension.

- **remote_apply** (*callable or None*) – If not None, `remote_apply(compilation_worker, kwargs)` must call *compilation_worker()* on any remote node. Typically, this might point to the *apply* method of an `ipyparallel` View instance. The *remote_apply* argument should only be given if *run_delayed()* will be called with an argument *map* that will push the calculation of a trajectory to a remote node.

Raises

- `ValueError` – if *executable* name or *qsd_lib* are invalid
- `subprocess.CalledProcessError` – if compilation fails

run (*seed=None*, *workdir=None*, *keep=False*, *delay=False*)

Run the QSD program. The *compile()* method must have been called before *run*. If *compile()* was called with `delay=True`, compile at this point and run the resulting program. Otherwise, just run the existing program from the earlier compilation. The resulting directory data is returned, and in addition the *traj_data* attribute is updated to include the new trajectories (in addition to any previous trajectories)

The *run* method may be called repeatedly to accumulate trajectories.

Parameters

- **seed** (*int*) – Random number generator seed (unsigned integer), will be passed to the executable as the only argument.
- **workdir** (*str or None*) – The directory in which to (temporarily) create the output files. If None, a temporary directory will be used. Otherwise, the *workdir* must exist. Environment variables and '~' will be expanded.
- **keep** (*bool*) – If True, keep QSD output files inside *workdir*.
- **delay** (*bool*) – If True, schedule the run to be performed at a later point in time, when the *run_delayed()* routine is called.

Returns Averaged data obtained from the newly simulated trajectories only. None if *delay=True*.

Return type *qnet.misc.trajectory_data.TrajectoryData*

Raises

- *QSDCodeGenError* – if *compile()* was not called
- `OSError` – if creating/removing files/folders fails
- `subprocess.CalledProcessError` – if delayed compilation fails or executable returns with non-zero exit code
- `ValueError` – if seed is not unique

Note: The only way to run multiple trajectories in parallel is by giving `delay=True`. After preparing an arbitrary number of trajectories by repeated calls to *run()*. Then *run_delayed()* must be called with a *map* argument that supports parallel execution.

---

run_delayed (*map=<class 'map'>*, *n_procs_extend=1*, *_run_worker=None*)

Execute all scheduled runs (see *delay* option in *run()* method), possibly in parallel.

Parameters

- **map** (*callable*) – map(qsd_run_worker, list_of_kwargs) must be equivalent to [qsd_run_worker(kwargs) for kwargs in list_of_kwargs]. Defaults to the builtin *map* routine, which will process the scheduled runs serially.
- **n_procs_extend** (*int*) – Number of local processes to use when averaging over trajectories.

Raises TypeError – If *map* does not return a list of *TrajectoryData* instances.

Note: Parallel execution is achieved by passing an appropriate *map* routine. For example, map=multiprocessing.Pool(5).map would use a local thread pool of 5 workers. Another alternative would be the *map* method of an ipyparallel View. If (and only if) the View connects remote IPython engines, *compile()* must have been called with an appropriate *remote_apply* argument that compiled the QSD program on all of the remote engines.

qnet.misc.qsd_codegen.**expand_cmd**(*cmd*)
Return a copy of the array *cmd*, where for each element of the *cmd* array, environment variables and '~' are expanded

qnet.misc.qsd_codegen.**compilation_worker**(*kwargs*, *_runner=None*)
Worker to perform compilation, suitable e.g. for being run on an IPython cluster. All arguments are in the *kwargs* dictionary.

Keys

- **executable** (*str*) – Name of the executable to be created. Nothing will be expanded.
- **path** (*str*) – Path where the executable should be created, as absolute path or relative to the current working directory. Environment variables and '~' will be expanded.
- **cc_code** (*str*) – Multiline string that contains the entire C++ program to be compiled
- **keep_cc** (*bool*) – Keep C++ file after compilation? It will have the same name as the executable, with an added .cc file extension.
- **cmd** (*list of str*) – Command line arguments (see *args* in *subprocess.check_output*). In each argument, environment variables are expanded, and '~' is expanded to $HOME. It must meet the following requirements:
  - the compiler (first argument) must be in the $PATH
  - Invocation of the command must compile a C++ file with the name *executable*.cc in the current working directory to *exectuable*, also in the current working directoy. It must *not* take into account *path*. This is because the working directory for the subprocess handling the command invocation will be set to *path*. Thus, that is where the executable will be created.

Returns Absolute path of the compiled executable

Raises

- subprocess.CalledProcessError – if compilation fails
- OSError – if creating/removing files/folder fails

qnet.misc.qsd_codegen.**qsd_run_worker**(*kwargs*, *_runner=None*)
Worker to perform run of a previously compiled program (see *compilation_worker()*), suitable e.g. for being run on an IPython cluster. All arguments are in the *kwargs* dictionary.

Keys

- **executable** (*str*) – Name of the executable to be run. Nothing will be expanded. This should generally be only the name of the executable, but it can also be a path relative to `kwargs['path']`, or a (fully expanded) absolute path, in which case `kwargs['path']` is ignored.
- **path** (*str*) – Path where the executable can be found, as absolute path or relative to the current working directory. Environment variables and '~' will be expanded.
- **seed** (*int*) – Seed (unsigned int) to be passed as argument to the executable
- **operators** (*dict or OrderedDict of str to str)*) – Mapping of operator name to filename, see *operators* parameter of `from_qsd_data()`
- **workdir** (*str or None*) – The working directory in which to execute the executable (relative to the current working directory). The output files defined in *operators* will be created in this folder. If None, a temporary directory will be used. If *workdir* does not exist yet, it will be created.
- **keep** (*bool*) – If True, keep the QSD output files. If False, remove the output files as well as any parent folders that may have been created alongside with *workdir*

Raises `FileNotFoundError` – if *executable* does not exist in *path*

Returns Expectation values and variances of the observables, from the newly simulated trajectories only (instance of `TrajectoryData`)

`qnet.misc.qsd_codegen.`**`sanitize_name`**(*name*, *allowed_letters*, *replacements*)

Return a sanitized *name*, where all letters that occur as keys in *replacements* are replaced by their corresponding values, and any letters that do not match *allowed_letters* are dropped

Parameters

- **`name`** (*`str`*) – string to be sanitized
- **`allowed_letters`** (*`regex`*) – compiled regular expression that any allowed letter must match
- **`replacement`** (*`dict of str to str`*) – dictionary of mappings

Returns sanitized name

Return type str

Example:

```
>>> sanitize_filename = partial(sanitize_name,
...         allowed_letters=re.compile(r'[.a-zA-Z0-9_-]'),
...         replacements={'^':'_', '+':'_', '*':'_', ' ':'_'})
>>> sanitize_filename.__doc__ = "Sanitize name to be used as a filename"
>>> sanitize_filename('\chi^{(1)}_1')
'chi_1_1'
```

qnet.misc.testing_tools module

Collection of routines needed for testing. This includes proto-fixtures, i.e. routines that should be imported and then turned into a fixture with the pytest.fixture decorator.

See <https://pytest.org/latest/fixture.html>

Summary

Functions:

datadir	Proto-fixture responsible for searching a folder with the same name of test module and, if available, moving all contents to a temporary directory so tests can use them freely.
fake_traj	Return a new trajectory that has the same data as *traj_template*, but a different *ID* and *seed*.
qsd_traj	Return a proto-fixture that returns a `TrajectoryData` instance based on all the `*.out` file in the given folder (relative to the test datadir), and with the given *seed*.

Reference

qnet.misc.testing_tools.**datadir**(*tmpdir*, *request*)

Proto-fixture responsible for searching a folder with the same name of test module and, if available, moving all contents to a temporary directory so tests can use them freely.

In any test, import the datadir routine and turn it into a fixture:

```
>>> import pytest
>>> import qnet.misc.testing_tools
>>> datadir = pytest.fixture(qnet.misc.testing_tools.datadir)
```

qnet.misc.testing_tools.**qsd_traj**(*datadir*, *folder*, *seed*)

Return a proto-fixture that returns a `TrajectoryData` instance based on all the `*.out` file in the given folder (relative to the test datadir), and with the given *seed*.

The returned function should be turned into a fixture:

```
>>> import pytest
>>> import qnet.misc.testing_tools
>>> from qnet.misc.testing_tools import qsd_traj
>>> datadir = pytest.fixture(qnet.misc.testing_tools.datadir)
>>> traj1 = pytest.fixture(qsd_traj(datadir, 'traj1', 102121))
```

qnet.misc.testing_tools.**fake_traj**(*traj_template*, *ID*, *seed*)

Return a new trajectory that has the same data as *traj_template*, but a different *ID* and *seed*. Assumes that *traj_template* only has a single record (i.e., it was created from QSD data)

qnet.misc.trajectory_data module

Summary

Exceptions:

TrajectoryParserError	Exception raised if a *TrajectoryData* file is malformed

Classes:

TrajectoryData	Tabular data of expectation values for one or more trajectories.

Reference

exception qnet.misc.trajectory_data.**TrajectoryParserError**
Bases: Exception

Exception raised if a *TrajectoryData* file is malformed

class qnet.misc.trajectory_data.**TrajectoryData**(*ID*, *dt*, *seed*, *n_trajectories*, *data*)
Bases: object

Tabular data of expectation values for one or more trajectories. Multiple *TrajectoryData* objects can be combined with the *extend()* method, in order to accumulate averages over an arbitrary number o trajectories. As much as possible, it is checked that all trajectories are statistically independent. A record is kept to ensure exact reproducibility.

Parameters

- **ID** (*str*) – A unique, RFC 4122 compliant identifier (as generated by *new_id()*)
- **dt** (*float*) – Time step between data points (>0)
- **seed** (*int*) – The random number generator seed on which the data is based
- **n_trajectories** (*int*) – The number of trajectories from which the data is averaged (It is assumed that the random number generator was seeded with the given seed, and then the given number of trajectories were calculated *sequentially*)
- **data** (*dict of str to tuple of arrays*) – dictionary (preferably *OrderedDict*) of expectation value data. The value of `data[operator_name]` must be a tuple of four numpy arrays (real part of expectation value, imaginary part of expectation value, real part of standard deviation, imaginary part of standard deviation). The operator names must contain only ASCII characters and must be shorter than `col_width - 10`.

Raises ValueError – if *ID* is not RFC 4122 compliant, *dt* is an invalid or non-positive float, or *data* does not follow the correct structure.

Attributes

- **ID** (*str*) – A unique ID for the current state of the *TrajectoryData* (read-only). See *ID* property.
- **table** (*OrderedDict of str to numpy array*) – A table that contains four column for every known operator (real/imaginary part of the expectation value, real/imaginary part of the variance). Note that the *table* attribute can easily be converted to a `pandas.DataFrame` (`DataFrame(data=traj.table)`). The *table* attribute should be considered read-only.
- **dt** (*float*) – Time step between data points
- **nt** (*int*) – Number of time steps / data points
- **operators** (*list of str*) – An iterator of the operator names. The column names in the *table* attribute derive from these. Assuming "X" is one of the operator names, there will be four keys in *table*: "Re[<X>]", "Im[<X>]", "Re[var(X)]", "Im[var(X)]"

- **record** (*OrderedDic of str to tuple of int, int, list*) – A copy of the complete record of how the averaged expectation values for all operators were obtained. See discussion of the `record` property.
- **col_width** (*int*) – width of the data columns when writing out data. Defaults to 25 (allowing to full double precision). Note that operator names may be at most of length `col_width-10`

col_width = 25

copy()
Return a (deep) copy of the current object

classmethod read(*filename*)
Read in TrajectoryData from the given filename. The file must be in the format generated by the *write* method.

Raises `TrajectoryParserError` – if the file has an incorrect format

classmethod from_qsd_data(*operators*, *seed*, *workdir='.'*)
Instantiate from one or more QSD output files specified as values of the dictionary *operators*

Each QSD output file must have the following structure:

- The first line must start with the string "Number_of_Trajectories", followed by an integer (separated by whitespace)
- All following lines must contain five floating point numbers (time, real/imaginary part of expectation value, and real/imaginary part of variance), separated by whitespace.

All QSD output files must contain the same number of lines, specify the same number of trajectories, and use the same time grid values (first column). It is the user's responsibility to ensure that all out output files were indeed generated in a single QSD run using the specified initial seed for the random number generator.

Parameters

- **operators** (`dict of str to str`) – dictionary (preferably *OrderedDict*) of operator name to filename. The filenames are relative to the *workdir*. Each filename must contain data in the format described above
- **seed** (`int`) – The seed to the random number generator that was used to produce the data file
- **workdir** (`str`) – directory to which the filenames in *operators* are relative to

Raises `ValueError` – if any of the data files do not have the correct format or are inconsistent

---

Note: Remember that is is vitally important that all quantum trajectories that go into an average are statistically independent. The `TrajectoryData` class tries as much as possible to ensure this, by refusing to combine identical IDs, or trajectories originating from the same seed. To this end, in the `from_qsd_data()` method, the ID of the instantiated object will depend uniquely on the collective data read from the QSD output files.

---

classmethod new_id(*name=None*)
Generate a new unique identifier, as a string. The identifier will be RFC 4122 compliant. If name is None, the resulting ID will be random. Otherwise, name must be a string that the ID will depend on. That is, calling *new_id* repeatedly with the same *name* will result in identical IDs.

ID

A unique RFC 4122 compliant identifier. The identifier changes whenever the class data is modified (via the `extend()` method). Two instances of *TrajectoryData* with the same ID are assumed to be identical

record

A copy of the full trajectory record, i.e., a history of calls to the `extend()` method. Its purpose is to ensure that the data is completely reproducible. This entails storing the seed to the random number generator for all sets of trajectories.

The record is an *OrderedDict* that maps the original ID of any *TrajectoryData* instance combined via `extend()` to a tuple `(seed, n_trajectories, ops)`, where `seed` is the seed to the random number generator that was used to calculate a specific set of trajectories (sequentially), `n_trajectories` are the number of trajectories in that dataset, and `ops` is a list of operator names for which expectation values were calculated. This may be the complete list of operators in the *operators* attribute, or a subset of those operators (Not all trajectories have to include data for all operators).

For example, let's assume we have a `QSDCodeGen` instance to set up for a QSD propagation. Two observables 'X1', 'X2', have been added to be written to file 'X1.out', and 'X2.out'. The `set_trajectories()` method has been called with `n_trajectories=10`, after which a call to `run()` with argument `seed=SEED1`, performed a sequential propagation of 10 trajectories, with the averaged expectation values written to the output files.

This data may now be read into a new *TrajectoryData* instance `traj` via the `from_qsd_data()` class method (with *seed=SEED1*). The newly created instance (with, let's say, `ID='8d102e4b-...'`) will have one entry in its record:

```
'8d102e4b-...': (SEED1, 10, ['X1', 'X2'])
```

Now, let's say we add a new observable 'A2' (output file 'A2.out') for the `QSDCodeGen` instance (in addition to the existing observables X1, X2), and call the `run()` method again, with a new seed `SEED2`. We then update `traj` with a call such as:

```
traj.extend(TrajectoryData.from_qsd_data(
    {'X1':'X1.out', 'X2':'X2.out', 'A2':'A2.out'}, SEED2)
```

The record will now have an additional entry, e.g.:

```
'd9831647-...': (SEED2, 10, ['X1', 'X2', 'A2'])
```

`traj.table` will contain the averaged expectation values (average over 20 trajectories for 'X1', 'X2', and 10 trajectories for 'A2'). The record tells use that to reproduce this table, 10 sequential trajectories starting from `SEED1` must be performed for X1, X2, followed by another 10 trajectories for X1, X2, A2 starting from `SEED2`.

operators

Iterator over all operators

record_IDs

Set of all IDs in the record

dt

Time step between data points

nt

Number of time steps / data points

shape

Tuple `(n_row, n_cols)` for the data in self.table. The time grid is included in the column count

record_seeds
Set of all random number generator seeds in the record

tgrid
Time grid, as numpy array

to_str(*show_rows=-1*)
Generate full string representation of the *TrajectoryData*

Parameters **show_rows** (*int*) – If given > 0, maximum number of data rows to show. If there are more rows, they will be indicated by an ellipsis (`...`)

Raises `ValueError` – if any operator name is too long to generate a label that fits in the limit given by the *col_width* class attribute

write(*filename*)
Write data to a text file. The *TrajectoryData* may later be restored by the *read* class method from the same file

n_trajectories(*operator*)
Return the total number of trajectories for the given operator

extend(**others*, ***kwargs*)
Extend data with data from one or more other *TrajectoryData* instances, averaging the expectation values. Equivalently to `traj1.extend(traj2)`, the syntax `traj1 += traj2` may be used.

Raises

- `ValueError` – if *data* in *self* and and any element of *others* are incompatible
- `TypeError` – if any *others* are not an instance of *TrajectoryData*

qnet.printing package

Printing system for QNET Expressions and related objects

Submodules:

qnet.printing.ascii module

Summary

Functions:

ascii	Return an ascii textual representation of the given object /

Module data:

qnet.printing.ascii.**AsciiPrinter**

Reference

qnet.printing.ascii.**ascii**(*expr*)
Return an ascii textual representation of the given object / expression

qnet.printing.base module

Provides the base class for Printers

Summary

Classes:

Printer	Base class for Printers (and default ASCII printer)

Reference

class qnet.printing.base.**Printer**

Bases: object

Base class for Printers (and default ASCII printer)

Attributes

- **head_repr_fmt** (*str*) – The format for representing expressions in the form `head(arg1, arg2, ..., key1=val1, key2=val2, ...)`. Uses formatting keys *head* (`expr.__class__.__name__`), *args* (rendered representation of `expr.args`), and *kwargs* (rendered representation of `expr.kwargs`). Used by *render_head_repr()*
- **identity_sym** (*str*) – Representation of the identity operator
- **circuit_identify_fmt** (*str*) – Format for the identity in a Circuit, parametrized by the number of channels, given as the formatting key *cdim*.
- **dagger_sym** (*str*) – Symbol representing a dagger
- **daggered_sym** (*str*) – Superscript version of *dagger_sym*
- **permutation_sym** (*str*) – The identifier of a Circuit permutation
- **pseudo_daggered_sym** (*str*) – Superscript representing a pseudo-dagger
- **pal_left** (*str*) – The symbol/string for a left parenthesis
- **par_right** (*str*) – The symbol/string for a right parenthesis
- **brak_left** (*str*) – The symbol/string for a left square bracket
- **brak_right** (*str*) – The symbol/string for a right square bracket
- **arg_sep** (*str*) – The string that should be used to separate rendered arguments in a list (usually a comma)
- **scalar_product_sym** (*str*) – Symbol to indicate a product between two scalars
- **tensor_sym** (*str*) – Symbol to indicate a tensor product
- **inner_product_sym** (*str*) – Symbol to indicate an inner product
- **op_product_sym** (*str*) – Symbol to indicate a product between two operators in the the same Hilbert space
- **circuit_series** (*str*) – Infix symbol for a series product of two circuits
- **circuit_concat_sym** (*str*) – Infix symbol for a concatenation of two circuits

- **circuit_inverse_fmt** (*str*) – Format for rendering the series-inverse of a circuit element. Receives a formatting key *operand* of the rendered operand circuit element
- **circuit_fb_fmt** (*str*) – Format for rendering a feedback circuit element. Receives the formatting keys *operand*, *output*, and *input* that are the rendered operand circuit element, the index of the ouput port (as a string), and the index of the input port to which the feedback connects (also as a string)
- **op_trace_fmt** (*str*) – Format for rendering a trace. Receives the formatting keys *operand* (the object being traced) and *space* (the rendered label of the Hilbert space that is being traced over)
- **null_space_proj_sym** (*str*) – The identifier for a nullspace projector
- **hilbert_space_fmt** (*str*) – Format for rendering a `HilbertSpace` object. Receives the formatting key *label* with the rendered label of the Hilbert space
- **matrix_left_sym** (*str*) – The symbol that marks the beginning of a matrix, for rendering a `Matrix` instance
- **matrix_right_sym** (*str*) – The symbol that marks the end of a matrix
- **matrix_row_left_sym** (*str*) – Symbol that marks beginning of row in matrix
- **matrix_row_right_sym** (*str*) – Symbol that marks end of row in matrix
- **matrix_col_sep_sym** (*str*) – Symbol that separates the values in different columns of a matrix
- **matrix_row_sep_sym** (*str*) – Symbol that separates the rows of a matrix
- **bra_fmt** (*str*) – Format for rendering a `Bra` instance. Receives the formatting keys *label* (the rendered label of the state) and *space* (the rendered label of the Hilbert space)
- **ket_fmt** (*str*) – Format for rendering a `Ket` instance. Receives the formatting keys *label* and *space*
- **ketbra_fmt** (*str*) – Format for rendering a `KetBra` instance. Receives the formatting keys *label_i*, *label_j*, and *space*, for the rendered label of the "left" and "right" state, and the Hilbert space
- **braket_fmt** (*str*) – Format for rendering a `BraKet` instance. Receives the formatting keys *label_i*, *label_j*, and *space*.
- **cc_string** (*str*) – String to indicate the complex conjugate (in a sum)

head_repr_fmt = '{head}({args}{kwargs})'

identity_sym = '1'

circuit_identity_fmt = 'cid({cdim})'

zero_sym = '0'

dagger_sym = 'H'

daggered_sym = '^H'

permutation_sym = 'Perm'

pseudo_daggered_sym = '^+'

par_left = '('

par_right = ')'

brak_left = '['

brak_right = ']'

arg_sep = ', '

scalar_product_sym = '*'

tensor_sym = '*'

inner_product_sym = '*'

op_product_sym = '*'

circuit_series_sym = '<<'

circuit_concat_sym = '+'

circuit_inverse_fmt = '[{operand}]^{{-1}}'

circuit_fb_fmt = '[{operand}]_{{{output}->{input}}}'

op_trace_fmt = 'tr_({space})[{operand}]'

null_space_proj_sym = 'P_Ker'

hilbert_space_fmt = 'H_{label}'

matrix_left_sym = '['

matrix_right_sym = ']'

matrix_row_left_sym = '['

matrix_row_right_sym = ']'

matrix_col_sep_sym = ', '

matrix_row_sep_sym = ', '

bra_fmt = '<{label}|_({space})'

ket_fmt = '|{label}>_({space})'

ketbra_fmt = '|{label_i}><{label_j}|_({space})'

braket_fmt = '<{label_i}|{label_j}>_({space})'

cc_string = 'c.c.'

op_hs_super_sub = 1

classmethod render(*expr: typing.Any*, *adjoint=False*) → str
Render an expression (or the adjoint of the expression)

classmethod register(*expr*, *rendered*)
Register a fixed rendered string for the given *expr* in an internal registry. As a result, any call to `render()` for *expr* will immediately return *rendered*

classmethod update_registry(*mapping*)
Call `register(key, val)` for every key-value pair in the *mapping dictionary*

classmethod del_registered_expr(*expr*)
Remove the registered *expr* from the registry (cf. `register_expr()`)

classmethod clear_registry()
Clear the registry

classmethod **render_head_repr** (*expr: typing.Any*, *sub_render=None*, *key_sub_render=None*) → str
Render a textual representation of *expr* using `head_repr_fmt`. Positional and keyword arguments are recursively rendered using *sub_render*, which defaults to *cls.render* by default. If desired, a different renderer may be used for keyword arguments by giving *key_sub_renderer*

Raises `AttributeError` – if *expr* is not an instance of `Expression`, or more specifically, if *expr* does not have *args* and *kwargs* (respectively *minimal_kwargs*) properties

classmethod **render_op** (*identifier: str*, *hs=None*, *dagger=False*, *args=None*, *superop=False*) → str
Render an operator

Parameters

- **identifier** (*str*) – Name of the operator (unrendered string)
- **hs** (`HilbertSpace`) – Hilbert space instance of the operator
- **dagger** (*bool*) – Whether or not to render the operator with a dagger
- **args** (*list*) – List of arguments for the operator (list of expressions). These will be rendered through the *render* method and appended to the rendered operator in parentheses
- **superop** (*bool*) – Flag to indicate whether the operator is a superoperator

classmethod **render_string** (*ascii_str: str*) → str
Render an unrendered (ascii) string, resolving e.g. greek letters and sub-/superscripts

classmethod **render_sum** (*operands*, *plus_sym='+'*, *minus_sym='-'*, *padding=' '*, *adjoint=False*)
Render a sum

classmethod **render_product** (*operands*, *prod_sym*, *sum_classes*, *minus_sym='-'*, *padding=' '*, *adjoint=False*, *dynamic_prod_sym=None*)
Render a product

classmethod **render_hs_label** (*hs: typing.Any*) → str
Render the total label for the given Hilbert space

classmethod **render_scalar** (*value: typing.Any*, *adjoint=False*) → str
Render a scalar *value* (numeric or symbolic)

qnet.printing.srepr module

Provides printers for a full-structured representation

Summary

Classes:

IndentedSReprPrinter	Printer for rendering an expression in such a way that the resulting

Functions:

srepr	Render the given expression into a string that can be evaluated in an appropriate context to re-instantiate an identical expression.

Module data:

`qnet.printing.srepr.`**`SReprPrinter`**

Reference

class `qnet.printing.srepr.`**`IndentedSReprPrinter`**(*indent=0*)

Bases: `qnet.printing.base.Printer`

Printer for rendering an expression in such a way that the resulting string can be evaluated in an appropriate context to re-instantiate an identical object, using nested indentation (implementing `srepr(expr, indented=True)`

`render`(*expr*, *adjoint=False*)

Render the given expression. Not that *adjoint* must be False

`render_sympy`(*expr*, *adjoint=False*)

Render a sympy expression

`render_numpy_matrix`(*expr*, *adjoint=False*)

`render_head_repr`(*expr: typing.Any*, *sub_render=None*, *key_sub_render=None*) → str

Render a multiline textual representation of *expr*

Raises `AttributeError` – if *expr* is not an instance of `Expression`, or more specifically, if *expr* does not have *args* and *kwargs* (respectively *minimal_kwargs*) properties

`qnet.printing.srepr.`**`srepr`**(*expr*, *indented=False*)

Render the given expression into a string that can be evaluated in an appropriate context to re-instantiate an identical expression. If *indented* is False (default), the resulting string is a single line. Otherwise, the result is a multiline string, and each positional and keyword argument of each *Expression* is on a separate line, recursively indented to produce a tree-like output.

See also:

qnet.printing.tree_str produces an output similar to *srepr* with `indented=True`. Unlike *srepr*, however, *tree_str* uses line drawings for the tree, shows arguments directly on the same line as the expression they belong to, and cannot be evaluated.

qnet.printing.tex module

Routines for rendering expressions to LaTeX

Summary

Functions:

`tex`	Return a LaTeX string representation of the given *expr*

Module data:

`qnet.printing.tex.`**`LaTeXPrinter`**

Reference

qnet.printing.tex.**tex**(*expr*)
Return a LaTeX string representation of the given *expr*

qnet.printing.tree module

Tree printer for Expressions

Summary

Functions:

`shorten_renderer`	Return a modified that returns the representation of expr, or '...' if
`tree`	Print a tree representation of the structure of *expr*
`tree_str`	Give the output of *tree* as a multiline string, using line drawings to

Module data:

qnet.printing.tree.**HeadStrPrinter**

Reference

qnet.printing.tree.**shorten_renderer**(*renderer*, *max_len*)
Return a modified that returns the representation of expr, or '...' if that representation is longer than *max_len*

qnet.printing.tree.**tree**(*expr*, *attr='operands'*, *padding=''*, *to_str=<bound method HeadStrPrinter.render of <class 'qnet.printing.tree.HeadStrPrinter'>>*, *exclude_type=None*, *depth=None*, *unicode=True*, *_last=False*, *_root=True*, *_level=0*, *_print=True*)
Print a tree representation of the structure of *expr*

Parameters

- **expr** (`Expression`) – expression to render
- **attr** (`str`) – The attribute from which to get the children of *expr*
- **padding** (`str`) – Whitespace by which the entire tree is idented
- **to_str** (`callable`) – Renderer for *expr*
- **exclude_type** (`type`) – Type (or list of types) which should never be expanded recursively
- **depth** (`int or None`) – Maximum depth of the tree to be printed
- **unicode** (`bool`) – If True, use unicode line-drawing symbols for the tree. If False, use an ASCII approximation

See also:

`tree_str()` return the result as a string, instead of printing it

`qnet.printing.tree.`**`tree_str`**`(`*expr*, ***kwargs*`)`

Give the output of *tree* as a multiline string, using line drawings to visualize the hierarchy of expressions (similar to the `tree` unix command line program for showing directory trees)

See also:

`qnet.printing.srepr()` with `indented=True` produces a similar tree-like rendering of the given expression that can be re-evaluated to the original expression.

qnet.printing.unicode module

Routines for rendering expressions to Unicode

Summary

Functions:

`unicode`	Return a unicode representation of the given *expr*
`unicode_sub_super`	Try to render a subscript string in unicode, fall back on ascii if this

Module data:

`qnet.printing.unicode.`**`UnicodePrinter`**

Reference

`qnet.printing.unicode.`**`unicode`**`(`*expr*`)`

Return a unicode representation of the given *expr*

`qnet.printing.unicode.`**`unicode_sub_super`**`(`*string*, *mapping*, *max_len=None*`)`

Try to render a subscript string in unicode, fall back on ascii if this is not possible

Summary

Functions:

`configure_printing`	context manager for temporarily changing the printing paremters. This
`init_printing`	Initialize printing

`__all__`: *`ascii`*, *`configure_printing`*, *`init_printing`*, *`srepr`*, *`tex`*, *`tree`*, *`unicode`*

Reference

`qnet.printing.`**`init_printing`**`(`*use_unicode=True*, *str_printer=None*, *repr_printer=None*, *cached_rendering=True*, *implicit_tensor=False*, *_init_sympy=True*`)`

Initialize printing

- Initialize *sympy* printing with the given *use_unicode* (i.e. call *sympy.init_printing*)
- Set the printers for textual representations (`str` and `repr`) of Expressions

•Configure whether *ascii*, *unicode*, and *tex* representations should be cached. If caching is enabled, the representations are rendered only once. This means that any configuration of the corresponding printers must be made before generating the representation for the first time.

Parameters

- **use_unicode** (*bool*) – If True, use unicode symbols. If False, restrict to ascii. Besides initializing *sympy* printing, this only determins the default *str* and *repr* printer. Thus, if *str_printer* and *repr_printer* are given, *use_unicode* has almost no effect.
- **str_printer** (Printer, *str*, *or None*) – The printer to be used for `str(expr)`. Must be an instance of `Printer` or one of the strings 'ascii', 'unicode', 'unicode', 'latex', or 'srepr', corresponding to *AsciiPrinter*, *UnicodePrinter*, *LaTeXPrinter*, and *SReprPrinter* respectively. If not given, either *AsciiPrinter* or *UnicodePrinter* is set, depending on *use_unicode*.
- **repr_printer** (Printer, *str*, *or None*) – Like *str_printer*, but for `repr(expr)`. This is also what is displayed in an interactive Python session
- **cached_rendering** (*bool*) – Flag whether the results of `ascii(expr)`, `unicode(expr)`, and `tex(expr)` should be cached
- **implicit_tensor** (*bool*) – If True, don't use tensor product symbols in the standard tex representation

Notes

•This routine does not set custom printers for rendering *ascii*, *unicode*, and *tex*. To use a non-default printer, you must assign directly to the corresponding class attributes of *Expression*.

•*str* and *repr* representations are never *directly* cached (but the printers they delegate to may use caching)

`qnet.printing.`**`configure_printing`**`(**kwargs)`

context manager for temporarily changing the printing paremters. This takes the same values as *init_printing*

qnet.qhdl package

Submodules:

qnet.qhdl.parser_QHDLParser_parsetab module

qnet.qhdl.qhdl module

This module contains the code to convert a circuit specified in QHDL into a Gough-James circuit expression.

The other module in this package `qhdl_parser` implements an actual parser for the qhdl source text, while this file then converts structured netlist information into a circuit expression.

For more details on the QHDL syntax, see *The QHDL Syntax*.

Summary

Exceptions:

QHDLError

Classes:

Architecture
BasicInterface
Component
Entity
QHDLObject

Functions:

dict_keys_sorted_by_val
gtype_compatible
my_debug

Reference

qnet.qhdl.qhdl.**my_debug**(*msg*)

exception qnet.qhdl.qhdl.**QHDLError**
Bases: Exception

class qnet.qhdl.qhdl.**QHDLObject**
Bases: object

to_python()

to_qhdl()

qnet.qhdl.qhdl.**gtype_compatible**(*c_t*, *g_t*)

class qnet.qhdl.qhdl.**BasicInterface**(*identifier*, *generics*, *ports*)
Bases: *qnet.qhdl.qhdl.QHDLObject*

to_qhdl(*tab_level*)

generics_to_qhdl(*tab_level*)

ports_to_qhdl(*tab_level*)

cid = 0

in_port_identifiers = []

out_port_identifiers = []

inout_port_identifiers = []

port_identifiers
The port_identifiers property.

generic_identifiers
The generic_identifiers property.

gids
The generic_identifiers property.

class qnet.qhdl.qhdl.**Entity**(*identifier*, *generics*, *ports*)
Bases: *qnet.qhdl.qhdl.BasicInterface*

to_qhdl(*tab_level=0*)

class qnet.qhdl.qhdl.**Component**(*identifier*, *generics*, *ports*)
Bases: *qnet.qhdl.qhdl.BasicInterface*

to_qhdl(*tab_level=0*)

qnet.qhdl.qhdl.**dict_keys_sorted_by_val**(*dd*)

class qnet.qhdl.qhdl.**Architecture**(*identifier*, *entity*, *components*, *signals*, *assignments*, *global_assignments={}*)
Bases: *qnet.qhdl.qhdl.QHDLObject*

signals = []

lossy_signals = []

global_inout = {}

global_out = {}

global_in = {}

inout_to_signal = {}

out_to_signal = {}

in_to_signal = {}

signal_to_global_in = {}

signal_to_global_out = {}

to_circuit(*identifier_postfix='‘*)
Compute a circuit algebra expression from the QHDL code and return the circuit expression, the all_symbols appearing in it and the component instance assignments

to_qhdl(*tab_level=0*)

qnet.qhdl.qhdl_parser module

The PLY-based QHDLParser class.

Summary

Classes:

QHDLParser

Reference

class qnet.qhdl.qhdl_parser.**QHDLParser**(***kw*)
Bases: *qnet.misc.parser.Parser*

parse(*inputstring*)

create_circuit_lib(*arch_id=None*)

reserved = {'generic': 'GENERIC', 'out': 'OUT', 'component': 'COMPONENT', 'fieldmode': 'FIELDMODE', 'real':

tokens = ['GENERIC', 'OUT', 'COMPONENT', 'FIELDMODE', 'REAL', 'OF', 'IN', 'MAP', 'IS', 'COMPLEX', 'EN

t_ignore = ' \t\x0c'

t_NEWLINE (*t*)
n+

t_ASSIGN = ':='

t_FEEDRIGHT = '=>'

t_FEEDLEFT = '<='

t_LPAREN = '\\('

t_RPAREN = '\\)'

t_COMMA = ','

t_SEMI = ';'

t_COLON = ':'

t_ID (*t*)
[_A-Za-z][w_]*

t_ICONST = '-?\\d+'

t_FCONST = '-?((\\d+)(\\.\\d+)(e(\\+|-)?(\\d+))? | (\\d+)e(\\+|-)?(\\d+))'

t_comment (*t*)
–[^n]*

t_error (*t*)

start = 'top_level_list'

p_top_level_list (*p*)

top_level_list [top_level_list top_level_unit]

top_level_unit

p_top_level_unit (*p*)

top_level_unit [entity_declaration]

architecture_declaration

p_entity_declaration (*p*)
entity_declaration : ENTITY ID IS generic_clause port_clause END opt_entity opt_id SEMI

p_opt_entity (*p*)

opt_entity [ENTITY]

empty

p_opt_id (*p*)

opt_id [ID]

empty

p_opt_semi (*p*)

opt_semi [SEMI]

empty

p_generic_clause (*p*)

generic_clause [generic_statement]

empty

p_empty (*p*)
empty :

p_generic_statement (*p*)
generic_statement : GENERIC LPAREN generic_list opt_semi RPAREN SEMI

p_generic_list (*p*)

generic_list [generic_list SEMI generic_entry_group]

generic_entry_group

p_generic_entry_group (*p*)
generic_entry_group : id_list COLON generic_type generic_default

p_id_list (*p*)

id_list [id_list COMMA ID]

ID

p_generic_type (*p*)
generic_type : REAL | COMPLEX | INT

p_generic_default (*p*)

generic_default [ASSIGN number]

empty

p_number (*p*)

number [simple_number]

complex

p_simple_number (*p*)

simple_number [int]

real

p_int (*p*)
int : ICONST

p_real (*p*)
real : FCONST

p_complex (*p*)
complex : LPAREN simple_number COMMA simple_number RPAREN

p_port_clause (*p*)

port_clause [port_statement]

empty

p_port_statement (*p*)
port_statement : PORT LPAREN port_list opt_semi RPAREN SEMI

p_port_list (*p*)

port_list [with_io_port_list]

non_io_port_list

p_with_io_port_list (*p*)

with_io_port_list [io_port_entry_group SEMI non_io_port_list]

io_port_entry_group

p_non_io_port_list (*p*)

non_io_port_list [non_io_port_entry_group SEMI non_io_port_list]

non_io_port_entry_group

p_non_io_port_entry_group (*p*)
non_io_port_entry_group : id_list COLON signal_direction signal_type

p_io_port_entry_group (*p*)
io_port_entry_group : id_list COLON INOUT signal_type

p_signal_direction (*p*)

signal_direction [IN]

OUT

p_signal_type (*p*)

signal_type [FIELDMODE]

LOSSY_FIELDMODE

p_architecture_declaration (*p*)
architecture_declaration : ARCHITECTURE ID OF ID IS architecture_head BEGIN instance_mapping_assignment_list feedleft_assignment_list END opt_arch opt_id SEMI

p_architecture_head (*p*)
architecture_head : component_declaration_list signal_list

p_opt_arch (*p*)

opt_arch [ARCHITECTURE]

empty

p_component_declaration_list (*p*)

component_declaration_list [component_declaration_list component_declaration]

component_declaration

p_component_declaration (*p*)
component_declaration : COMPONENT ID generic_clause port_clause END COMPONENT opt_id SEMI

p_signal_list (*p*)

signal_list [signal_list signal_entry_group]

signal_entry_group

p_signal_entry_group (*p*)
signal_entry_group : SIGNAL id_list COLON signal_type SEMI

p_instance_mapping_assignment_list (*p*)

instance_mapping_assignment_list [instance_mapping_assignment_list instance_mapping_assignment]

instance_mapping_assignment

p_instance_mapping_assignment (*p*)
instance_mapping_assignment : ID COLON ID generic_map port_map

p_generic_map (*p*)

generic_map [GENERIC MAP LPAREN feedright_generic_assignment_list RPAREN SEMI]

empty

p_feedright_generic_assignment_list (*p*)

feedright_generic_assignment_list [feedright_generic_assignment_list COMMA feedright_generic_assignment]

feedright_generic_assignment

p_id_or_value (*p*)

id_or_value [ID]

number

p_feedright_generic_assignment (*p*)

feedright_generic_assignment [ID FEEDRIGHT id_or_value]

id_or_value

p_feedright_port_assignment_list (*p*)

feedright_port_assignment_list [feedright_port_assignment_list COMMA feedright_port_assignment]

feedright_port_assignment

p_feedright_port_assignment (*p*)

feedright_port_assignment [ID FEEDRIGHT ID]

ID

p_port_map (*p*)

port_map [PORT MAP LPAREN feedright_port_assignment_list RPAREN SEMI]

empty

p_feedleft_assignment_list (*p*)

feedleft_assignment_list [feedleft_assignment_list feedleft_assignment]

feedleft_assignment

p_feedleft_assignment (*p*)

feedleft_assignment [ID FEEDLEFT ID SEMI]

empty

p_error (*p*)

`__all__`: *init_printing*

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Bibliography

[GoughJames08] Gough & James (2008). Quantum Feedback Networks: Hamiltonian Formulation. Communications in Mathematical Physics, 287(3), 1109-1132. doi:10.1007/s00220-008-0698-8

[GoughJames09] Gough & James (2009). The Series Product and Its Application to Quantum Feedforward and Feedback Networks. IEEE Transactions on Automatic Control, 54(11), 2530-2544. doi:10.1109/TAC.2009.2031205

[QHDL] Tezak, N., Niederberger, A., Pavlichin, D. S., Sarma, G., & Mabuchi, H. (2012). Specification of photonic circuits using quantum hardware description language. Philosophical transactions A, 370(1979), 5270–90. doi:10.1098/rsta.2011.0526

[Mabuchi11] Mabuchi, H. (2011). Nonlinear interferometry approach to photonic sequential logic. Appl. Phys. Lett. 99, 153103 (2011), doi:10.1063/1.3650250

Python Module Index

q

qnet, 55
qnet.algebra, 56
qnet.algebra.abstract_algebra, 56
qnet.algebra.circuit_algebra, 62
qnet.algebra.hilbert_space_algebra, 74
qnet.algebra.matrix_algebra, 77
qnet.algebra.operator_algebra, 81
qnet.algebra.ordering, 91
qnet.algebra.pattern_matching, 92
qnet.algebra.permutations, 96
qnet.algebra.singleton, 99
qnet.algebra.state_algebra, 100
qnet.algebra.super_operator_algebra, 105
qnet.circuit_components, 111
qnet.circuit_components.and_cc, 112
qnet.circuit_components.beamsplitter_cc, 113
qnet.circuit_components.component, 113
qnet.circuit_components.delay_cc, 115
qnet.circuit_components.displace_cc, 115
qnet.circuit_components.double_sided_jaynes_cummings_cc, 116
qnet.circuit_components.double_sided_opo_cc, 117
qnet.circuit_components.inverting_fanout_cc, 118
qnet.circuit_components.kerr_cavity_cc, 119
qnet.circuit_components.latch_cc, 120
qnet.circuit_components.library, 121
qnet.circuit_components.linear_cavity_cc, 123
qnet.circuit_components.mach_zehnder_cc, 124
qnet.circuit_components.open_lossy_cc, 125
qnet.circuit_components.phase_cc, 126
qnet.circuit_components.pseudo_nand_cc, 126
qnet.circuit_components.pseudo_nand_latch_cc, 127
qnet.circuit_components.relay_cc, 127
qnet.circuit_components.relay_double_probe_cc, 128
qnet.circuit_components.single_sided_jaynes_cumming 129
qnet.circuit_components.single_sided_opo_cc, 131
qnet.circuit_components.three_port_kerr_cavity_cc, 131
qnet.circuit_components.three_port_opo_cc, 134
qnet.circuit_components.two_port_kerr_cavity_cc, 135
qnet.circuit_components.z_probe_cavity_cc, 136
qnet.convert, 137
qnet.convert.to_qutip, 137
qnet.convert.to_sympy_matrix, 138
qnet.misc, 139
qnet.misc.circuit_visualization, 139
qnet.misc.euler_mayurama, 141
qnet.misc.kerr_model_matrices, 141
qnet.misc.parse_circuit_strings, 143
qnet.misc.parser, 145
qnet.misc.qsd_codegen, 145
qnet.misc.testing_tools, 152
qnet.misc.trajectory_data, 153
qnet.printing, 157
qnet.printing.ascii, 157
qnet.printing.base, 158
qnet.printing.srepr, 161
qnet.printing.tex, 162
qnet.printing.tree, 163
qnet.printing.unicode, 164
qnet.qhdl, 165
qnet.qhdl.qhdl, 165
qnet.qhdl.qhdl_parser, 167

Index

Symbols

__getstate__() (qnet.algebra.abstract_algebra.Expression method), 59
__iter__() (qnet.misc.qsd_codegen.QSDOperator method), 147
__len__() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
__ne__() (qnet.algebra.abstract_algebra.Expression method), 58
__str__() (qnet.misc.qsd_codegen.QSDOperator method), 147

A

ABCD (class in qnet.algebra.circuit_algebra), 67
act_locally() (in module qnet.algebra.state_algebra), 105
act_locally_times_tensor() (in module qnet.algebra.state_algebra), 105
add_observable() (qnet.misc.qsd_codegen.QSDCodeGen method), 148
Adjoint (class in qnet.algebra.operator_algebra), 89
adjoint() (in module qnet.algebra.operator_algebra), 91
adjoint() (qnet.algebra.matrix_algebra.Matrix method), 79
adjoint() (qnet.algebra.operator_algebra.Operator method), 83
adjoint() (qnet.algebra.state_algebra.Bra method), 104
adjoint() (qnet.algebra.state_algebra.Ket method), 101
AlgebraError, 57
AlgebraException, 57
all_symbols() (in module qnet.algebra.abstract_algebra), 59
all_symbols() (qnet.algebra.abstract_algebra.Expression method), 58
all_symbols() (qnet.algebra.circuit_algebra.CircuitSymbol method), 68
all_symbols() (qnet.algebra.circuit_algebra.SLH method), 66
all_symbols() (qnet.algebra.hilbert_space_algebra.LocalSpace method), 77
all_symbols() (qnet.algebra.matrix_algebra.Matrix method), 79
all_symbols() (qnet.algebra.operator_algebra.Displace method), 87
all_symbols() (qnet.algebra.operator_algebra.LocalOperator method), 84
all_symbols() (qnet.algebra.operator_algebra.OperatorSymbol method), 84
all_symbols() (qnet.algebra.operator_algebra.OperatorTrace method), 88
all_symbols() (qnet.algebra.operator_algebra.Phase method), 87
all_symbols() (qnet.algebra.operator_algebra.ScalarTimesOperator method), 88
all_symbols() (qnet.algebra.operator_algebra.Squeeze method), 87
all_symbols() (qnet.algebra.state_algebra.CoherentStateKet method), 102
all_symbols() (qnet.algebra.state_algebra.KetSymbol method), 102
all_symbols() (qnet.algebra.state_algebra.LocalKet method), 102
all_symbols() (qnet.algebra.super_operator_algebra.SuperOperatorSymbol method), 107
all_symbols() (qnet.circuit_components.component.SubComponent method), 115
alpha (qnet.circuit_components.displace_cc.Displace attribute), 116
alpha (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
alpha (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
alpha (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
alpha (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
alpha (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
ampl (qnet.algebra.state_algebra.CoherentStateKet attribute), 102

And (class in qnet.circuit_components.and_cc), 112
anti_commutator() (in module qnet.algebra.super_operator_algebra), 109
Architecture (class in qnet.qhdl.qhdl), 167
arg_sep (qnet.printing.base.Printer attribute), 160
args (qnet.algebra.abstract_algebra.Expression attribute), 58
args (qnet.algebra.abstract_algebra.Operation attribute), 59
args (qnet.algebra.circuit_algebra.ABCD attribute), 68
args (qnet.algebra.circuit_algebra.CircuitSymbol attribute), 68
args (qnet.algebra.circuit_algebra.CPermutation attribute), 69
args (qnet.algebra.circuit_algebra.SLH attribute), 66
args (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 76
args (qnet.algebra.matrix_algebra.Matrix attribute), 78
args (qnet.algebra.operator_algebra.Displace attribute), 87
args (qnet.algebra.operator_algebra.LocalOperator attribute), 84
args (qnet.algebra.operator_algebra.LocalSigma attribute), 87
args (qnet.algebra.operator_algebra.OperatorSymbol attribute), 84
args (qnet.algebra.operator_algebra.Phase attribute), 87
args (qnet.algebra.operator_algebra.Squeeze attribute), 87
args (qnet.algebra.state_algebra.CoherentStateKet attribute), 102
args (qnet.algebra.state_algebra.KetSymbol attribute), 102
args (qnet.algebra.super_operator_algebra.SuperOperatorSymbol attribute), 107
args (qnet.circuit_components.component.Component attribute), 114
args (qnet.circuit_components.component.SubComponent attribute), 114
ascii() (in module qnet.printing.ascii), 157
AsciiPrinter (in module qnet.printing.ascii), 157
assoc() (in module qnet.algebra.abstract_algebra), 59

B

B1 (qnet.circuit_components.and_cc.And attribute), 112
B1 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
B1 (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
B11 (qnet.circuit_components.latch_cc.Latch attribute), 121
B12 (qnet.circuit_components.latch_cc.Latch attribute), 121
B2 (qnet.circuit_components.and_cc.And attribute), 112
B2 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
B2 (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
B21 (qnet.circuit_components.latch_cc.Latch attribute), 121
B22 (qnet.circuit_components.latch_cc.Latch attribute), 121
B3 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
B3 (qnet.circuit_components.latch_cc.Latch attribute), 121
BadLiouvillianError, 106
BadPermutationError, 97
BasicInterface (class in qnet.qhdl.qhdl), 166
basis (qnet.algebra.hilbert_space_algebra.HilbertSpace attribute), 76
basis (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 76
basis (qnet.algebra.hilbert_space_algebra.ProductSpace attribute), 77
basis_state() (in module qnet.convert.to_sympy_matrix), 139
BasisKet (class in qnet.algebra.state_algebra), 102
BasisNotSetError, 75
Beamsplitter (class in qnet.circuit_components.beamsplitter_cc), 113
beta (qnet.circuit_components.latch_cc.Latch attribute), 121
beta (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
block_matrix() (in module qnet.algebra.matrix_algebra), 79
block_perm_and_perms_within_blocks() (in module qnet.algebra.permutations), 99
block_perms (qnet.algebra.circuit_algebra.CPermutation attribute), 69
block_structure (qnet.algebra.circuit_algebra.Circuit attribute), 64
block_structure (qnet.algebra.matrix_algebra.Matrix attribute), 78
Bra (class in qnet.algebra.state_algebra), 104
bra (qnet.algebra.state_algebra.BraKet attribute), 104
bra (qnet.algebra.state_algebra.KetBra attribute), 105
bra_fmt (qnet.printing.base.Printer attribute), 160
brak_left (qnet.printing.base.Printer attribute), 159
brak_right (qnet.printing.base.Printer attribute), 159
BraKet (class in qnet.algebra.state_algebra), 104
braket_fmt (qnet.printing.base.Printer attribute), 160
BS (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 126
BS1 (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127

BS2 (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127

C

C (qnet.circuit_components.and_cc.And attribute), 112
C (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
C1 (qnet.circuit_components.latch_cc.Latch attribute), 121
C2 (qnet.circuit_components.latch_cc.Latch attribute), 121
cache_attr() (in module qnet.algebra.abstract_algebra), 57
camelcase_to_underscore() (in module qnet.circuit_components.library), 123
CannotConvertToABCD, 64
CannotConvertToSLH, 64
CannotEliminateAutomatically, 64
CannotSimplify, 57
CannotSymbolicallyDiagonalize, 106
CannotVisualize, 64
CavityPort (class in qnet.circuit_components.double_sided_jaynes_cummings_cc), 117
CavityPort (class in qnet.circuit_components.linear_cavity_cc), 124
CavityPort (class in qnet.circuit_components.single_sided_jaynes_cummings_cc), 130
cc_string (qnet.printing.base.Printer attribute), 160
cdim (qnet.algebra.circuit_algebra.ABCD attribute), 68
cdim (qnet.algebra.circuit_algebra.Circuit attribute), 64
cdim (qnet.algebra.circuit_algebra.CircuitSymbol attribute), 68
cdim (qnet.algebra.circuit_algebra.Concatenation attribute), 70
cdim (qnet.algebra.circuit_algebra.CPermutation attribute), 69
cdim (qnet.algebra.circuit_algebra.Feedback attribute), 70
cdim (qnet.algebra.circuit_algebra.SeriesInverse attribute), 71
cdim (qnet.algebra.circuit_algebra.SeriesProduct attribute), 69
cdim (qnet.algebra.circuit_algebra.SLH attribute), 66
CDIM (qnet.circuit_components.and_cc.And attribute), 112
CDIM (qnet.circuit_components.beamsplitter_cc.Beamsplitter attribute), 113
CDIM (qnet.circuit_components.component.Component attribute), 114
cdim (qnet.circuit_components.component.Component attribute), 114
cdim (qnet.circuit_components.component.SubComponent attribute), 114
CDIM (qnet.circuit_components.delay_cc.Delay attribute), 115
CDIM (qnet.circuit_components.displace_cc.Displace attribute), 116
CDIM (qnet.circuit_components.double_sided_jaynes_cummings_cc.Doub attribute), 116
CDIM (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
CDIM (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
CDIM (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
CDIM (qnet.circuit_components.latch_cc.Latch attribute), 121
CDIM (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
CDIM (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
CDIM (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
CDIM (qnet.circuit_components.phase_cc.Phase attribute), 126
CDIM (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 126
CDIM (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDLatc attribute), 127
CDIM (qnet.circuit_components.relay_cc.Relay attribute), 128
CDIM (qnet.circuit_components.relay_double_probe_cc.RelayDoubleProbe attribute), 129
CDIM (qnet.circuit_components.single_sided_jaynes_cummings_cc.Single attribute), 130
CDIM (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
CDIM (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrC attribute), 133
CDIM (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
CDIM (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCav attribute), 135
CDIM (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 136
check_cdims() (in module qnet.algebra.circuit_algebra), 64
check_idempotent_create() (in module qnet.algebra.abstract_algebra), 59
check_kets_same_space() (in module qnet.algebra.state_algebra), 101
check_op_ket_space() (in module qnet.algebra.state_algebra), 101
check_permutation() (in module qnet.algebra.permutations), 97
chi (qnet.circuit_components.and_cc.And attribute), 112
chi (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
chi (qnet.circuit_components.kerr_cavity_cc.KerrCavity

attribute), 120
chi (qnet.circuit_components.latch_cc.Latch attribute), 121
chi (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
chi (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
chi (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
chi (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 135
cid (qnet.qhdl.qhdl.BasicInterface attribute), 166
cid() (in module qnet.algebra.circuit_algebra), 71
cid_1 (in module qnet.algebra.circuit_algebra), 63
CIdentity (in module qnet.algebra.circuit_algebra), 63
Circuit (class in qnet.algebra.circuit_algebra), 64
circuit_concat_sym (qnet.printing.base.Printer attribute), 160
circuit_fb_fmt (qnet.printing.base.Printer attribute), 160
circuit_identity() (in module qnet.algebra.circuit_algebra), 71
circuit_identity_fmt (qnet.printing.base.Printer attribute), 159
circuit_inverse_fmt (qnet.printing.base.Printer attribute), 160
circuit_series_sym (qnet.printing.base.Printer attribute), 160
CircuitSymbol (class in qnet.algebra.circuit_algebra), 68
CircuitZero (in module qnet.algebra.circuit_algebra), 63
clear_registry() (qnet.printing.base.Printer class method), 160
coeff (qnet.algebra.operator_algebra.ScalarTimesOperator attribute), 88
coeff (qnet.algebra.state_algebra.ScalarTimesKet attribute), 103
coeff (qnet.algebra.super_operator_algebra.ScalarTimesSuperOperator attribute), 108
coherent_input() (qnet.algebra.circuit_algebra.Circuit method), 66
CoherentStateKet (class in qnet.algebra.state_algebra), 102
col_width (qnet.misc.trajectory_data.TrajectoryData attribute), 155
commutator() (in module qnet.algebra.super_operator_algebra), 109
compilation_worker() (in module qnet.misc.qsd_codegen), 151
compile() (qnet.misc.qsd_codegen.QSDCodeGen method), 149
compile_cmd (qnet.misc.qsd_codegen.QSDCodeGen attribute), 148
Component (class in qnet.circuit_components.component), 114
Component (class in qnet.qhdl.qhdl), 167
compose_permutations() (in module qnet.algebra.permutations), 98
concatenate_permutations() (in module qnet.algebra.permutations), 98
concatenate_slh() (qnet.algebra.circuit_algebra.SLH method), 67
Concatenation (class in qnet.algebra.circuit_algebra), 69
configure_printing() (in module qnet.printing), 165
conjugate() (qnet.algebra.matrix_algebra.Matrix method), 79
conjugate() (qnet.algebra.operator_algebra.Operator method), 83
connect() (in module qnet.algebra.circuit_algebra), 74
convert_to_qutip() (in module qnet.convert.to_qutip), 137
convert_to_spaces() (in module qnet.algebra.hilbert_space_algebra), 75
convert_to_sympy_matrix() (in module qnet.convert.to_sympy_matrix), 139
copy() (qnet.misc.trajectory_data.TrajectoryData method), 155
CPermutation (class in qnet.algebra.circuit_algebra), 68
Create (class in qnet.algebra.operator_algebra), 84
create() (qnet.algebra.abstract_algebra.Expression class method), 58
create() (qnet.algebra.circuit_algebra.CPermutation class method), 69
create() (qnet.algebra.circuit_algebra.Feedback class method), 70
create() (qnet.algebra.circuit_algebra.SeriesInverse class method), 71
create() (qnet.algebra.hilbert_space_algebra.ProductSpace class method), 77
create() (qnet.algebra.state_algebra.TensorKet class method), 103
create() (qnet.algebra.super_operator_algebra.SuperOperatorTimes class method), 108
create_circuit_lib() (qnet.qhdl.qhdl_parser.QHDLParser method), 167
create_operator_pm_cc() (in module qnet.algebra.operator_algebra), 91
creduce() (qnet.algebra.circuit_algebra.Circuit method), 65

D

dag (qnet.algebra.state_algebra.Bra attribute), 104
dag (qnet.algebra.state_algebra.Ket attribute), 101
dag() (qnet.algebra.matrix_algebra.Matrix method), 79
dag() (qnet.algebra.operator_algebra.Operator method), 83
dagger_sym (qnet.printing.base.Printer attribute), 159
daggered_sym (qnet.printing.base.Printer attribute), 159
datadir() (in module qnet.misc.testing_tools), 153
DecayChannel (class in qnet.circuit_components.double_sided_jaynes_cummings_cc),

117
DecayChannel (class in qnet.circuit_components.single_sided_jaynes_cummings_cc), 130
decompose_space() (in module qnet.algebra.operator_algebra), 90
del_registered_expr() (qnet.printing.base.Printer class method), 160
Delay (class in qnet.circuit_components.delay_cc), 115
delegate_to_method (qnet.algebra.circuit_algebra.Feedback attribute), 70
delegate_to_method (qnet.algebra.circuit_algebra.SeriesInverse attribute), 71
delegate_to_method() (in module qnet.algebra.operator_algebra), 83
Delta (qnet.circuit_components.and_cc.And attribute), 112
Delta (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
Delta (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
Delta (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
Delta (qnet.circuit_components.latch_cc.Latch attribute), 121
Delta (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
Delta (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
Delta (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
Delta (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
Delta (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
Delta (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
Delta (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 135
Delta (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 136
Delta_a (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
Delta_a (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
Delta_f (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
Delta_f (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
Destroy (class in qnet.algebra.operator_algebra), 85
diagm() (in module qnet.algebra.matrix_algebra), 79
dict_keys_sorted_by_val() (in module qnet.qhdl.qhdl), 167
diff() (qnet.algebra.operator_algebra.Operator method), 83
dimension (qnet.algebra.hilbert_space_algebra.HilbertSpace attribute), 76
dimension (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 77
dimension (qnet.algebra.hilbert_space_algebra.ProductSpace attribute), 77
DisjunctCommutativeHSOrder (class in qnet.algebra.ordering), 92
Displace (class in qnet.algebra.operator_algebra), 87
Displace (class in qnet.circuit_components.displace_cc), 116
DoubleSidedJaynesCummings (class in qnet.circuit_components.double_sided_jaynes_cummings_cc), 116
DoubleSidedOPO (class in qnet.circuit_components.double_sided_opo_cc), 118
draw_circuit() (in module qnet.misc.circuit_visualization), 140
draw_circuit_canvas() (in module qnet.misc.circuit_visualization), 139
dt (qnet.misc.trajectory_data.TrajectoryData attribute), 156

E

element_wise() (qnet.algebra.matrix_algebra.Matrix method), 79
empty_trivial() (in module qnet.algebra.hilbert_space_algebra), 75
Entity (class in qnet.qhdl.qhdl), 166
euler_mayurama_complex() (in module qnet.misc.euler_mayurama), 141
eval_adiabatic_limit() (in module qnet.algebra.circuit_algebra), 74
expand() (qnet.algebra.circuit_algebra.SLH method), 67
expand() (qnet.algebra.matrix_algebra.Matrix method), 79
expand() (qnet.algebra.operator_algebra.Operator method), 83
expand() (qnet.algebra.state_algebra.Ket method), 101
expand() (qnet.algebra.super_operator_algebra.SuperOperator method), 106
expand_cmd() (in module qnet.misc.qsd_codegen), 151
expand_operator_pm_cc() (in module qnet.algebra.operator_algebra), 91
expr_order_key() (in module qnet.algebra.ordering), 92
Expression (class in qnet.algebra.abstract_algebra), 58
extend() (qnet.misc.trajectory_data.TrajectoryData method), 157
extended_arg_patterns() (qnet.algebra.pattern_matching.Pattern method), 95
extra_binary_rules() (in module qnet.algebra.abstract_algebra), 61

extra_rules() (in module qnet.algebra.abstract_algebra), 61
extract_signal() (in module qnet.algebra.circuit_algebra), 72
extract_signal_circuit() (in module qnet.algebra.circuit_algebra), 72

F

factor_coeff() (in module qnet.algebra.operator_algebra), 91
factor_for_space() (qnet.algebra.operator_algebra.OperatorTimes method), 88
factor_for_space() (qnet.algebra.state_algebra.TensorKet method), 103
factor_for_trace() (in module qnet.algebra.operator_algebra), 90
fake_traj() (in module qnet.misc.testing_tools), 153
FB() (in module qnet.algebra.circuit_algebra), 71
Feedback (class in qnet.algebra.circuit_algebra), 70
feedback() (qnet.algebra.circuit_algebra.Circuit method), 65
FeedbackPort (class in qnet.circuit_components.z_probe_cavity_cc), 137
filter_neutral() (in module qnet.algebra.abstract_algebra), 60
find_kets() (in module qnet.misc.qsd_codegen), 146
findall() (qnet.algebra.pattern_matching.Pattern method), 95
FOCK_DIM (qnet.circuit_components.delay_cc.Delay attribute), 115
FOCK_DIM (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
FOCK_DIM (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
FOCK_DIM (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
FOCK_DIM (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
FOCK_DIM (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
FOCK_DIM (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
FOCK_DIM (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
FOCK_DIM (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
FOCK_DIM (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 136
fock_space (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
fock_space (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
from_qsd_data() (qnet.misc.trajectory_data.TrajectoryData class method), 155
full_block_perm() (in module qnet.algebra.permutations), 99
FullCommutativeHSOrder (class in qnet.algebra.ordering), 92
FullSpace (in module qnet.algebra.hilbert_space_algebra), 75

G

g (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSide attribute), 117
g (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSided attribute), 130
gamma (qnet.circuit_components.double_sided_jaynes_cummings_cc.Doub attribute), 117
gamma (qnet.circuit_components.single_sided_jaynes_cummings_cc.Single attribute), 130
gamma (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 136
gamma_p (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 136
generate_code() (qnet.misc.qsd_codegen.QSDCodeGen method), 149
generic_identifiers (qnet.qhdl.qhdl.BasicInterface attribute), 166
generics_to_qhdl() (qnet.qhdl.qhdl.BasicInterface method), 166
get_basis() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
get_blocks() (qnet.algebra.circuit_algebra.Circuit method), 65
get_coeffs() (in module qnet.algebra.operator_algebra), 90
get_common_block_structure() (in module qnet.algebra.circuit_algebra), 71
get_dimension() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
get_observable() (qnet.misc.qsd_codegen.QSDCodeGen method), 148
getABCD() (in module qnet.algebra.circuit_algebra), 73
getCDIM() (in module qnet.circuit_components.library), 123
gids (qnet.qhdl.qhdl.BasicInterface attribute), 166
global_in (qnet.qhdl.qhdl.Architecture attribute), 167
global_inout (qnet.qhdl.qhdl.Architecture attribute), 167
global_out (qnet.qhdl.qhdl.Architecture attribute), 167
gtype_compatible() (in module qnet.qhdl.qhdl), 166

H

H (qnet.algebra.matrix_algebra.Matrix attribute), 79
has_minus_prefactor() (qnet.algebra.operator_algebra.ScalarTimesOperator static method), 88
head_repr_fmt (qnet.printing.base.Printer attribute), 159

HeadStrPrinter (in module qnet.printing.tree), 163
hilbert_space_fmt (qnet.printing.base.Printer attribute), 160
HilbertSpace (class in qnet.algebra.hilbert_space_algebra), 75
hstackm() (in module qnet.algebra.matrix_algebra), 79

I

ID (qnet.misc.trajectory_data.TrajectoryData attribute), 155
idem() (in module qnet.algebra.abstract_algebra), 60
identifier (qnet.algebra.operator_algebra.LocalOperator attribute), 84
identity_matrix() (in module qnet.algebra.matrix_algebra), 80
identity_sym (qnet.printing.base.Printer attribute), 159
IdentityOperator (in module qnet.algebra.operator_algebra), 82
IdentitySuperOperator (in module qnet.algebra.super_operator_algebra), 106
II (in module qnet.algebra.operator_algebra), 82
Im() (in module qnet.algebra.matrix_algebra), 80
ImAdjoint() (in module qnet.algebra.matrix_algebra), 80
implied_local_space() (in module qnet.algebra.operator_algebra), 82
in_port_identifiers (qnet.qhdl.qhdl.BasicInterface attribute), 166
in_to_signal (qnet.qhdl.qhdl.Architecture attribute), 167
IncompatibleBlockStructures, 64
IndentedSReprPrinter (class in qnet.printing.srepr), 162
index_in_block() (qnet.algebra.circuit_algebra.Circuit method), 65
init_printing() (in module qnet.printing), 164
inner_product_sym (qnet.printing.base.Printer attribute), 160
inout_port_identifiers (qnet.qhdl.qhdl.BasicInterface attribute), 166
inout_to_signal (qnet.qhdl.qhdl.Architecture attribute), 167
instance_caching (qnet.algebra.abstract_algebra.Expression attribute), 58
instantiation (qnet.misc.qsd_codegen.QSDOperator attribute), 147
instantiator (qnet.misc.qsd_codegen.QSDOperator attribute), 146
intersect() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 75
intersect() (qnet.algebra.hilbert_space_algebra.LocalSpace method), 77
intersect() (qnet.algebra.hilbert_space_algebra.ProductSpace method), 77
invert_permutation() (in module qnet.algebra.permutations), 97
InvertingFanout (class in qnet.circuit_components.inverting_fanout_cc), 119
is_strict_subfactor_of() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
is_strict_subfactor_of() (qnet.algebra.hilbert_space_algebra.LocalSpace method), 77
is_strict_subfactor_of() (qnet.algebra.hilbert_space_algebra.ProductSpace method), 77
is_strict_tensor_factor_of() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
is_tensor_factor_of() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76
is_zero (qnet.algebra.matrix_algebra.Matrix attribute), 78
isdisjoint() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 76

J

Jminus (class in qnet.algebra.operator_algebra), 86
Jmjmcoeff() (in module qnet.algebra.operator_algebra), 91
Jpjmcoeff() (in module qnet.algebra.operator_algebra), 91
Jplus (class in qnet.algebra.operator_algebra), 85
Jz (class in qnet.algebra.operator_algebra), 85
Jzjmcoeff() (in module qnet.algebra.operator_algebra), 91

K

K (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
kappa (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
kappa (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
kappa (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleS attribute), 130
kappa (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
kappa_1 (qnet.circuit_components.and_cc.And attribute), 112
kappa_1 (qnet.circuit_components.double_sided_jaynes_cummings_cc.Dou attribute), 117
kappa_1 (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
kappa_1 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
kappa_1 (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
kappa_1 (qnet.circuit_components.latch_cc.Latch attribute), 121
kappa_1 (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124

kappa_1 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
kappa_1 (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
kappa_1 (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 135
kappa_2 (qnet.circuit_components.and_cc.And attribute), 112
kappa_2 (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
kappa_2 (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
kappa_2 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
kappa_2 (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
kappa_2 (qnet.circuit_components.latch_cc.Latch attribute), 121
kappa_2 (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
kappa_2 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
kappa_2 (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
kappa_2 (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 136
kappa_3 (qnet.circuit_components.and_cc.And attribute), 112
kappa_3 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
kappa_3 (qnet.circuit_components.latch_cc.Latch attribute), 121
kappa_3 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
kappa_3 (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
KC (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 126
KerrCavity (class in qnet.circuit_components.kerr_cavity_cc), 120
KerrPort (class in qnet.circuit_components.kerr_cavity_cc), 120
KerrPort (class in qnet.circuit_components.three_port_kerr_cavity_cc), 133
KerrPort (class in qnet.circuit_components.two_port_kerr_cavity_cc), 136
Ket (class in qnet.algebra.state_algebra), 101
ket (qnet.algebra.state_algebra.Bra attribute), 104
ket (qnet.algebra.state_algebra.BraKet attribute), 104
ket (qnet.algebra.state_algebra.KetBra attribute), 105
ket (qnet.algebra.state_algebra.OperatorTimesKet attribute), 104
ket_fmt (qnet.printing.base.Printer attribute), 160
KetBra (class in qnet.algebra.state_algebra), 105
ketbra_fmt (qnet.printing.base.Printer attribute), 160
KetPlus (class in qnet.algebra.state_algebra), 102
KetSymbol (class in qnet.algebra.state_algebra), 102
KeyTuple (class in qnet.algebra.ordering), 92
known_types (qnet.misc.qsd_codegen.QSDCodeGen attribute), 148
known_types (qnet.misc.qsd_codegen.QSDOperator attribute), 146
kwargs (qnet.algebra.abstract_algebra.Expression attribute), 58
kwargs (qnet.algebra.circuit_algebra.Feedback attribute), 70
kwargs (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 77
kwargs (qnet.algebra.operator_algebra.LocalOperator attribute), 84
kwargs (qnet.algebra.operator_algebra.OperatorPlusMinusCC attribute), 89
kwargs (qnet.algebra.operator_algebra.OperatorSymbol attribute), 84
kwargs (qnet.algebra.operator_algebra.OperatorTrace attribute), 88
kwargs (qnet.algebra.state_algebra.CoherentStateKet attribute), 102
kwargs (qnet.algebra.state_algebra.KetSymbol attribute), 102
kwargs (qnet.algebra.super_operator_algebra.SuperOperatorSymbol attribute), 107
kwargs (qnet.circuit_components.component.Component attribute), 114

L

label (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 76
label (qnet.algebra.state_algebra.Bra attribute), 104
label (qnet.algebra.state_algebra.KetSymbol attribute), 102
label (qnet.algebra.state_algebra.TensorKet attribute), 103
label (qnet.algebra.super_operator_algebra.SuperOperatorSymbol attribute), 107
Latch (class in qnet.circuit_components.latch_cc), 121
LaTeXPrinter (in module qnet.printing.tex), 162
lindblad() (in module qnet.algebra.super_operator_algebra), 109
LinearCavity (class in qnet.circuit_components.linear_cavity_cc), 124
liouvillian() (in module qnet.algebra.super_operator_algebra), 110
liouvillian_normal_form() (in module qnet.algebra.super_operator_algebra), 110
local_factors (qnet.algebra.hilbert_space_algebra.HilbertSpace attribute), 75

local_factors (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 77
local_factors (qnet.algebra.hilbert_space_algebra.ProductSpace attribute), 77
local_ops() (in module qnet.misc.qsd_codegen), 146
LocalKet (class in qnet.algebra.state_algebra), 102
LocalOperator (class in qnet.algebra.operator_algebra), 84
LocalProjector() (in module qnet.algebra.operator_algebra), 89
LocalSigma (class in qnet.algebra.operator_algebra), 87
LocalSpace (class in qnet.algebra.hilbert_space_algebra), 76
LossPort (class in qnet.circuit_components.z_probe_cavity_cc), 137
lossy_signals (qnet.qhdl.qhdl.Architecture attribute), 167
Ls (qnet.algebra.circuit_algebra.SLH attribute), 66
LSS_ci_ls (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 126

M

m (qnet.algebra.circuit_algebra.ABCD attribute), 68
MachZehnder (class in qnet.circuit_components.mach_zehnder_cc), 125
make_namespace_string() (in module qnet.circuit_components.library), 123
map_signals() (in module qnet.algebra.circuit_algebra), 72
map_signals_circuit() (in module qnet.algebra.circuit_algebra), 72
match() (qnet.algebra.pattern_matching.Pattern method), 95
match_pattern() (in module qnet.algebra.pattern_matching), 96
match_replace() (in module qnet.algebra.abstract_algebra), 60
match_replace_binary() (in module qnet.algebra.abstract_algebra), 61
MatchDict (class in qnet.algebra.pattern_matching), 93
Matrix (class in qnet.algebra.matrix_algebra), 78
matrix (qnet.algebra.matrix_algebra.Matrix attribute), 78
matrix_col_sep_sym (qnet.printing.base.Printer attribute), 160
matrix_left_sym (qnet.printing.base.Printer attribute), 160
matrix_right_sym (qnet.printing.base.Printer attribute), 160
matrix_row_left_sym (qnet.printing.base.Printer attribute), 160
matrix_row_right_sym (qnet.printing.base.Printer attribute), 160
matrix_row_sep_sym (qnet.printing.base.Printer attribute), 160
minimal_kwargs (qnet.algebra.abstract_algebra.Expression attribute), 58
minimal_kwargs (qnet.algebra.hilbert_space_algebra.LocalSpace attribute), 77
minimal_kwargs (qnet.algebra.operator_algebra.LocalOperator attribute), 84
minimal_kwargs (qnet.algebra.operator_algebra.OperatorPlusMinusCC attribute), 89
model_matrices() (in module qnet.misc.kerr_model_matrices), 142
model_matrices_complex() (in module qnet.misc.kerr_model_matrices), 143
model_matrices_symbolic() (in module qnet.misc.kerr_model_matrices), 143
move_drive_to_H() (in module qnet.algebra.circuit_algebra), 73
my_debug() (in module qnet.qhdl.qhdl), 166

N

n (qnet.algebra.circuit_algebra.ABCD attribute), 68
N (qnet.circuit_components.delay_cc.Delay attribute), 115
n_trajectories() (qnet.misc.trajectory_data.TrajectoryData method), 157
name (qnet.algebra.circuit_algebra.CircuitSymbol attribute), 68
name (qnet.circuit_components.component.Component attribute), 114
name (qnet.circuit_components.component.SubComponent attribute), 114
name (qnet.misc.qsd_codegen.QSDOperator attribute), 146
NAND1 (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDLa attribute), 127
NAND2 (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDLa attribute), 127
neutral_element (qnet.algebra.circuit_algebra.Concatenation attribute), 70
neutral_element (qnet.algebra.circuit_algebra.SeriesProduct attribute), 69
neutral_element (qnet.algebra.hilbert_space_algebra.ProductSpace attribute), 77
neutral_element (qnet.algebra.operator_algebra.OperatorPlus attribute), 87
neutral_element (qnet.algebra.operator_algebra.OperatorTimes attribute), 88
neutral_element (qnet.algebra.state_algebra.KetPlus attribute), 103
neutral_element (qnet.algebra.state_algebra.TensorKet attribute), 103
neutral_element (qnet.algebra.super_operator_algebra.SuperOperatorPlus attribute), 107
neutral_element (qnet.algebra.super_operator_algebra.SuperOperatorTimes attribute), 108

new_id() (qnet.misc.trajectory_data.TrajectoryData class method), 155
no_instance_caching() (in module qnet.algebra.abstract_algebra), 61
no_rules() (in module qnet.algebra.abstract_algebra), 61
NonSquareMatrix, 78
nt (qnet.misc.trajectory_data.TrajectoryData attribute), 156
null_space_proj_sym (qnet.printing.base.Printer attribute), 160
NullSpaceProjector (class in qnet.algebra.operator_algebra), 89

O

observable_names (qnet.misc.qsd_codegen.QSDCodeGen attribute), 148
observables (qnet.misc.qsd_codegen.QSDCodeGen attribute), 148
one_or_more (qnet.algebra.pattern_matching.Pattern attribute), 95
op (qnet.algebra.super_operator_algebra.SuperOperatorTimesOperator attribute), 109
op_hs_super_sub (qnet.printing.base.Printer attribute), 160
op_product_sym (qnet.printing.base.Printer attribute), 160
op_trace_fmt (qnet.printing.base.Printer attribute), 160
OpenLossy (class in qnet.circuit_components.open_lossy_cc), 125
operand (qnet.algebra.circuit_algebra.Feedback attribute), 70
operand (qnet.algebra.circuit_algebra.SeriesInverse attribute), 71
operand (qnet.algebra.operator_algebra.OperatorTrace attribute), 88
operand (qnet.algebra.operator_algebra.SingleOperatorOperation attribute), 84
operand (qnet.algebra.state_algebra.Bra attribute), 104
operand (qnet.algebra.super_operator_algebra.SuperAdjoint attribute), 108
operands (qnet.algebra.abstract_algebra.Operation attribute), 59
Operation (class in qnet.algebra.abstract_algebra), 59
Operator (class in qnet.algebra.operator_algebra), 83
operator (qnet.algebra.state_algebra.OperatorTimesKet attribute), 104
OperatorOperation (class in qnet.algebra.operator_algebra), 84
OperatorPlus (class in qnet.algebra.operator_algebra), 87
OperatorPlusMinusCC (class in qnet.algebra.operator_algebra), 89
operators (qnet.misc.trajectory_data.TrajectoryData attribute), 156
OperatorSymbol (class in qnet.algebra.operator_algebra), 84
OperatorTimes (class in qnet.algebra.operator_algebra), 88
OperatorTimesKet (class in qnet.algebra.state_algebra), 103
OperatorTrace (class in qnet.algebra.operator_algebra), 88
OPOPort (class in qnet.circuit_components.double_sided_opo_cc), 118
OPOPort (class in qnet.circuit_components.three_port_opo_cc), 135
order_key (qnet.algebra.operator_algebra.OperatorPlus attribute), 88
order_key (qnet.algebra.operator_algebra.OperatorTimes attribute), 88
order_key (qnet.algebra.state_algebra.KetPlus attribute), 103
order_key (qnet.algebra.state_algebra.TensorKet attribute), 103
order_key (qnet.algebra.super_operator_algebra.SuperOperatorPlus attribute), 107
order_key (qnet.algebra.super_operator_algebra.SuperOperatorTimes attribute), 108
order_key() (qnet.algebra.hilbert_space_algebra.ProductSpace class method), 77
orderby() (in module qnet.algebra.abstract_algebra), 60
out_in_pair (qnet.algebra.circuit_algebra.Feedback attribute), 70
out_port_identifiers (qnet.qhdl.qhdl.BasicInterface attribute), 166
out_to_signal (qnet.qhdl.qhdl.Architecture attribute), 167
OverlappingSpaces, 101

P

P (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
P (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
p_architecture_declaration() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_architecture_head() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_complex() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_component_declaration() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_component_declaration_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_empty() (qnet.qhdl.qhdl_parser.QHDLParser method), 169

p_entity_declaration() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_error() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedleft_assignment() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedleft_assignment_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedright_generic_assignment() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedright_generic_assignment_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedright_port_assignment() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_feedright_port_assignment_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_generic_clause() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_generic_default() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_generic_entry_group() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_generic_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_generic_map() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_generic_statement() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_generic_type() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_id_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_id_or_value() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_instance_mapping_assignment() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_instance_mapping_assignment_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_int() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_io_port_entry_group() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_non_io_port_entry_group() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_non_io_port_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_number() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_opt_arch() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_opt_entity() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_opt_id() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_opt_semi() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_port_clause() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_port_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_port_map() (qnet.qhdl.qhdl_parser.QHDLParser method), 171
p_port_statement() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_real() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
P_sigma() (in module qnet.algebra.circuit_algebra), 71
p_signal_direction() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_signal_entry_group() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_signal_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_signal_type() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
p_simple_number() (qnet.qhdl.qhdl_parser.QHDLParser method), 169
p_top_level_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_top_level_unit() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
p_with_io_port_list() (qnet.qhdl.qhdl_parser.QHDLParser method), 170
pad_with_identity() (in module qnet.algebra.circuit_algebra), 72
par_left (qnet.printing.base.Printer attribute), 159
par_right (qnet.printing.base.Printer attribute), 159
parent_component (qnet.circuit_components.component.SubComponent attribute), 114
parse() (qnet.misc.parser.Parser method), 145
parse() (qnet.qhdl.qhdl_parser.QHDLParser method), 167
parse_circuit_strings() (in module qnet.misc.parse_circuit_strings), 144
parse_file() (qnet.misc.parser.Parser method), 145
ParseCircuitStringError, 144
Parser (class in qnet.misc.parser), 145
ParsingError, 145
Pattern (class in qnet.algebra.pattern_matching), 93
pattern() (in module qnet.algebra.pattern_matching), 95
pattern_head() (in module

qnet.algebra.pattern_matching), 95
permutation (qnet.algebra.circuit_algebra.CPermutation attribute), 69
permutation_from_block_permutations() (in module qnet.algebra.permutations), 98
permutation_from_disjoint_cycles() (in module qnet.algebra.permutations), 97
permutation_matrix() (in module qnet.algebra.matrix_algebra), 80
permutation_sym (qnet.printing.base.Printer attribute), 159
permutation_to_block_permutations() (in module qnet.algebra.permutations), 97
permutation_to_disjoint_cycles() (in module qnet.algebra.permutations), 97
permute() (in module qnet.algebra.permutations), 98
Phase (class in qnet.algebra.operator_algebra), 86
Phase (class in qnet.circuit_components.phase_cc), 126
Phase1 (qnet.circuit_components.and_cc.And attribute), 112
Phase1 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
Phase1 (qnet.circuit_components.latch_cc.Latch attribute), 121
Phase2 (qnet.circuit_components.and_cc.And attribute), 112
Phase2 (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
Phase2 (qnet.circuit_components.latch_cc.Latch attribute), 121
Phase3 (qnet.circuit_components.latch_cc.Latch attribute), 121
phi (qnet.circuit_components.and_cc.And attribute), 112
phi (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
phi (qnet.circuit_components.latch_cc.Latch attribute), 121
phi (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
phi (qnet.circuit_components.phase_cc.Phase attribute), 126
phi (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
phip (qnet.circuit_components.and_cc.And attribute), 112
phip (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
phip (qnet.circuit_components.latch_cc.Latch attribute), 121
port1 (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
port1 (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
port1 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
port1 (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavit attribute), 136
port2 (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
port2 (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
port2 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCa attribute), 133
port2 (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavit attribute), 136
port3 (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCa attribute), 133
port_identifiers (qnet.qhdl.qhdl.BasicInterface attribute), 166
ports_to_qhdl() (qnet.qhdl.qhdl.BasicInterface method), 166
PORTSIN (qnet.circuit_components.and_cc.And attribute), 112
PORTSIN (qnet.circuit_components.beamsplitter_cc.Beamsplitter attribute), 113
PORTSIN (qnet.circuit_components.component.Component attribute), 114
PORTSIN (qnet.circuit_components.component.SubComponent attribute), 114
PORTSIN (qnet.circuit_components.delay_cc.Delay attribute), 115
PORTSIN (qnet.circuit_components.displace_cc.Displace attribute), 116
PORTSIN (qnet.circuit_components.double_sided_jaynes_cummings_cc.D attribute), 117
PORTSIN (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOP attribute), 118
PORTSIN (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
PORTSIN (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
PORTSIN (qnet.circuit_components.latch_cc.Latch attribute), 121
PORTSIN (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
PORTSIN (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
PORTSIN (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
PORTSIN (qnet.circuit_components.phase_cc.Phase attribute), 126
PORTSIN (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
PORTSIN (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDL attribute), 127
PORTSIN (qnet.circuit_components.relay_cc.Relay attribute), 128
PORTSIN (qnet.circuit_components.relay_double_probe_cc.RelayDoubleP

attribute), 129
PORTSIN (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
PORTSIN (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
PORTSIN (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
PORTSIN (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
PORTSIN (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 135
PORTSIN (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 136
PORTSOUT (qnet.circuit_components.and_cc.And attribute), 112
PORTSOUT (qnet.circuit_components.beamsplitter_cc.Beamsplitter attribute), 113
PORTSOUT (qnet.circuit_components.component.Component attribute), 114
PORTSOUT (qnet.circuit_components.component.SubComponent attribute), 114
PORTSOUT (qnet.circuit_components.delay_cc.Delay attribute), 115
PORTSOUT (qnet.circuit_components.displace_cc.Displace attribute), 116
PORTSOUT (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
PORTSOUT (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
PORTSOUT (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
PORTSOUT (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
PORTSOUT (qnet.circuit_components.latch_cc.Latch attribute), 121
PORTSOUT (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
PORTSOUT (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
PORTSOUT (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
PORTSOUT (qnet.circuit_components.phase_cc.Phase attribute), 126
PORTSOUT (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
PORTSOUT (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDLatch attribute), 127
PORTSOUT (qnet.circuit_components.relay_cc.Relay attribute), 128
PORTSOUT (qnet.circuit_components.relay_double_probe_cc.RelayDoubleProbe attribute), 129
PORTSOUT (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
PORTSOUT (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
PORTSOUT (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCavity attribute), 133
PORTSOUT (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 134
PORTSOUT (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavity attribute), 135
PORTSOUT (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 137
precedence (qnet.misc.parser.Parser attribute), 145
prepare_adiabatic_limit() (in module qnet.algebra.circuit_algebra), 73
prepare_sde() (in module qnet.misc.kerr_model_matrices), 143
Printer (class in qnet.printing.base), 158
ProbePort (class in qnet.circuit_components.z_probe_cavity_cc), 137
ProductSpace (class in qnet.algebra.hilbert_space_algebra), 77
ProtoExpr (class in qnet.algebra.pattern_matching), 96
pseudo_daggered_sym (qnet.printing.base.Printer attribute), 159
pseudo_inverse() (qnet.algebra.operator_algebra.Operator method), 83
PseudoInverse (class in qnet.algebra.operator_algebra), 89
PseudoNAND (class in qnet.circuit_components.pseudo_nand_cc), 126
PseudoNANDLatch (class in qnet.circuit_components.pseudo_nand_latch_cc), 127

Q

QHDLError, 166
QHDLObject (class in qnet.qhdl.qhdl), 166
QHDLParser (class in qnet.qhdl.qhdl_parser), 167
qnet (module), 55
qnet.algebra (module), 56
qnet.algebra.abstract_algebra (module), 56
qnet.algebra.circuit_algebra (module), 62
qnet.algebra.hilbert_space_algebra (module), 74
qnet.algebra.matrix_algebra (module), 77
qnet.algebra.operator_algebra (module), 81
qnet.algebra.ordering (module), 91
qnet.algebra.pattern_matching (module), 92
qnet.algebra.permutations (module), 96
qnet.algebra.singleton (module), 99
qnet.algebra.state_algebra (module), 100
qnet.algebra.super_operator_algebra (module), 105
qnet.circuit_components (module), 111
qnet.circuit_components.and_cc (module), 112
qnet.circuit_components.beamsplitter_cc (module), 113
qnet.circuit_components.component (module), 113

qnet.circuit_components.delay_cc (module), 115
qnet.circuit_components.displace_cc (module), 115
qnet.circuit_components.double_sided_jaynes_cummings_cc (module), 116
qnet.circuit_components.double_sided_opo_cc (module), 117
qnet.circuit_components.inverting_fanout_cc (module), 118
qnet.circuit_components.kerr_cavity_cc (module), 119
qnet.circuit_components.latch_cc (module), 120
qnet.circuit_components.library (module), 121
qnet.circuit_components.linear_cavity_cc (module), 123
qnet.circuit_components.mach_zehnder_cc (module), 124
qnet.circuit_components.open_lossy_cc (module), 125
qnet.circuit_components.phase_cc (module), 126
qnet.circuit_components.pseudo_nand_cc (module), 126
qnet.circuit_components.pseudo_nand_latch_cc (module), 127
qnet.circuit_components.relay_cc (module), 127
qnet.circuit_components.relay_double_probe_cc (module), 128
qnet.circuit_components.single_sided_jaynes_cummings_cc (module), 129
qnet.circuit_components.single_sided_opo_cc (module), 131
qnet.circuit_components.three_port_kerr_cavity_cc (module), 131
qnet.circuit_components.three_port_opo_cc (module), 134
qnet.circuit_components.two_port_kerr_cavity_cc (module), 135
qnet.circuit_components.z_probe_cavity_cc (module), 136
qnet.convert (module), 137
qnet.convert.to_qutip (module), 137
qnet.convert.to_sympy_matrix (module), 138
qnet.misc (module), 139
qnet.misc.circuit_visualization (module), 139
qnet.misc.euler_mayurama (module), 141
qnet.misc.kerr_model_matrices (module), 141
qnet.misc.parse_circuit_strings (module), 143
qnet.misc.parser (module), 145
qnet.misc.qsd_codegen (module), 145
qnet.misc.testing_tools (module), 152
qnet.misc.trajectory_data (module), 153
qnet.printing (module), 157
qnet.printing.ascii (module), 157
qnet.printing.base (module), 158
qnet.printing.srepr (module), 161
qnet.printing.tex (module), 162
qnet.printing.tree (module), 163
qnet.printing.unicode (module), 164
qnet.qhdl (module), 165
qnet.qhdl.qhdl (module), 165
qnet.qhdl.qhdl_parser (module), 167
qsd_run_worker() (in module qnet.misc.qsd_codegen), 151
qsd_traj() (in module qnet.misc.testing_tools), 153
qsd_type (qnet.misc.qsd_codegen.QSDOperator attribute), 146
QSDCCodePrinter (class in qnet.misc.qsd_codegen), 146
QSDCodeGen (class in qnet.misc.qsd_codegen), 147
QSDCodeGenError, 147
QSDOperator (class in qnet.misc.qsd_codegen), 146

R

Re() (in module qnet.algebra.matrix_algebra), 80
read() (qnet.misc.trajectory_data.TrajectoryData class method), 155
ReAdjoint() (in module qnet.algebra.matrix_algebra), 81
record (qnet.misc.trajectory_data.TrajectoryData attribute), 156
record_IDs (qnet.misc.trajectory_data.TrajectoryData attribute), 156
record_seeds (qnet.misc.trajectory_data.TrajectoryData attribute), 156
register() (qnet.printing.base.Printer class method), 160
Relay (class in qnet.circuit_components.relay_cc), 128
RelayControl (class in qnet.circuit_components.relay_cc), 128
RelayControl (class in qnet.circuit_components.relay_double_probe_cc), 129
RelayDoubleProbe (class in qnet.circuit_components.relay_double_probe_cc), 129
RelayOut (class in qnet.circuit_components.relay_cc), 128
RelayOut (class in qnet.circuit_components.relay_double_probe_cc), 129
remove() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 75
remove() (qnet.algebra.hilbert_space_algebra.LocalSpace method), 77
remove() (qnet.algebra.hilbert_space_algebra.ProductSpace method), 77
render() (qnet.algebra.circuit_algebra.Circuit method), 65
render() (qnet.printing.base.Printer class method), 160
render() (qnet.printing.srepr.IndentedSReprPrinter method), 162
render_head_repr() (qnet.printing.base.Printer class method), 160
render_head_repr() (qnet.printing.srepr.IndentedSReprPrinter method), 162
render_hs_label() (qnet.printing.base.Printer class method), 161

render_numpy_matrix() (qnet.printing.srepr.IndentedSReprPrinter method), 162
render_op() (qnet.printing.base.Printer class method), 161
render_product() (qnet.printing.base.Printer class method), 161
render_scalar() (qnet.printing.base.Printer class method), 161
render_string() (qnet.printing.base.Printer class method), 161
render_sum() (qnet.printing.base.Printer class method), 161
render_sympy() (qnet.printing.srepr.IndentedSReprPrinter method), 162
reserved (qnet.qhdl.qhdl_parser.QHDLParser attribute), 167
run() (qnet.misc.qsd_codegen.QSDCodeGen method), 150
run_delayed() (qnet.misc.qsd_codegen.QSDCodeGen method), 150

S

sanitize_name() (in module qnet.misc.qsd_codegen), 152
scalar_free_symbols() (in module qnet.algebra.operator_algebra), 91
scalar_product_sym (qnet.printing.base.Printer attribute), 160
ScalarTimesKet (class in qnet.algebra.state_algebra), 103
ScalarTimesOperator (class in qnet.algebra.operator_algebra), 88
ScalarTimesSuperOperator (class in qnet.algebra.super_operator_algebra), 108
series_expand() (qnet.algebra.matrix_algebra.Matrix method), 79
series_expand() (qnet.algebra.operator_algebra.Operator method), 83
series_inverse() (qnet.algebra.circuit_algebra.Circuit method), 65
series_with_permutation() (qnet.algebra.circuit_algebra.CPermutation method), 69
series_with_slh() (qnet.algebra.circuit_algebra.SLH method), 67
SeriesInverse (class in qnet.algebra.circuit_algebra), 70
SeriesProduct (class in qnet.algebra.circuit_algebra), 69
set_moving_basis() (qnet.misc.qsd_codegen.QSDCodeGen method), 148
set_trajectories() (qnet.misc.qsd_codegen.QSDCodeGen method), 148
set_union() (in module qnet.algebra.abstract_algebra), 59
shape (qnet.algebra.matrix_algebra.Matrix attribute), 78
shape (qnet.misc.trajectory_data.TrajectoryData attribute), 156
shorten_renderer() (in module qnet.printing.tree), 163
show() (qnet.algebra.circuit_algebra.Circuit method), 65
signal_to_global_in (qnet.qhdl.qhdl.Architecture attribute), 167
signal_to_global_out (qnet.qhdl.qhdl.Architecture attribute), 167
signals (qnet.qhdl.qhdl.Architecture attribute), 167
signature (qnet.algebra.hilbert_space_algebra.ProductSpace attribute), 77
simplify() (in module qnet.algebra.abstract_algebra), 59
simplify() (qnet.algebra.abstract_algebra.Expression method), 58
simplify_scalar() (in module qnet.algebra.operator_algebra), 90
simplify_scalar() (qnet.algebra.circuit_algebra.SLH method), 67
simplify_scalar() (qnet.algebra.matrix_algebra.Matrix method), 79
simplify_scalar() (qnet.algebra.operator_algebra.Operator method), 83
simplify_scalar() (qnet.algebra.super_operator_algebra.SuperOperator method), 107
single (qnet.algebra.pattern_matching.Pattern attribute), 95
SingleOperatorOperation (class in qnet.algebra.operator_algebra), 84
SingleSidedJaynesCummings (class in qnet.circuit_components.single_sided_jaynes_cummings_cc), 130
SingleSidedOPO (class in qnet.circuit_components.single_sided_opo_cc), 131
Singleton (class in qnet.algebra.singleton), 100
singleton_object() (in module qnet.algebra.singleton), 100
SLH (class in qnet.algebra.circuit_algebra), 66
SLH_to_qutip() (in module qnet.convert.to_qutip), 138
sop (qnet.algebra.super_operator_algebra.SuperOperatorTimesOperator attribute), 109
space (qnet.algebra.circuit_algebra.ABCD attribute), 68
space (qnet.algebra.circuit_algebra.Circuit attribute), 66
space (qnet.algebra.circuit_algebra.CircuitSymbol attribute), 68
space (qnet.algebra.circuit_algebra.Concatenation attribute), 70
space (qnet.algebra.circuit_algebra.CPermutation attribute), 69
space (qnet.algebra.circuit_algebra.Feedback attribute), 70
space (qnet.algebra.circuit_algebra.SeriesInverse attribute), 71
space (qnet.algebra.circuit_algebra.SeriesProduct attribute), 69
space (qnet.algebra.circuit_algebra.SLH attribute), 66
space (qnet.algebra.matrix_algebra.Matrix attribute), 79

space (qnet.algebra.operator_algebra.LocalOperator attribute), 84
space (qnet.algebra.operator_algebra.Operator attribute), 83
space (qnet.algebra.operator_algebra.OperatorOperation attribute), 84
space (qnet.algebra.operator_algebra.OperatorSymbol attribute), 84
space (qnet.algebra.operator_algebra.OperatorTrace attribute), 88
space (qnet.algebra.operator_algebra.ScalarTimesOperator attribute), 88
space (qnet.algebra.operator_algebra.SingleOperatorOperation attribute), 84
space (qnet.algebra.state_algebra.Bra attribute), 104
space (qnet.algebra.state_algebra.BraKet attribute), 105
space (qnet.algebra.state_algebra.Ket attribute), 101
space (qnet.algebra.state_algebra.KetBra attribute), 105
space (qnet.algebra.state_algebra.KetPlus attribute), 103
space (qnet.algebra.state_algebra.KetSymbol attribute), 102
space (qnet.algebra.state_algebra.OperatorTimesKet attribute), 104
space (qnet.algebra.state_algebra.ScalarTimesKet attribute), 103
space (qnet.algebra.state_algebra.TensorKet attribute), 103
space (qnet.algebra.super_operator_algebra.ScalarTimesSuperOperator attribute), 108
space (qnet.algebra.super_operator_algebra.SPost attribute), 109
space (qnet.algebra.super_operator_algebra.SPre attribute), 109
space (qnet.algebra.super_operator_algebra.SuperOperator attribute), 106
space (qnet.algebra.super_operator_algebra.SuperOperatorOperation attribute), 107
space (qnet.algebra.super_operator_algebra.SuperOperatorSymbol attribute), 107
space (qnet.algebra.super_operator_algebra.SuperOperatorTimesOperator attribute), 109
space (qnet.circuit_components.and_cc.And attribute), 112
space (qnet.circuit_components.component.Component attribute), 114
space (qnet.circuit_components.component.SubComponent attribute), 115
space (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
space (qnet.circuit_components.double_sided_opo_cc.DoubleSidedOPO attribute), 118
space (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
space (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
space (qnet.circuit_components.latch_cc.Latch attribute), 121
space (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
space (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
space (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 126
space (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
space (qnet.circuit_components.pseudo_nand_latch_cc.PseudoNANDLatch attribute), 127
space (qnet.circuit_components.relay_cc.Relay attribute), 128
space (qnet.circuit_components.relay_double_probe_cc.RelayDoubleProbe attribute), 129
space (qnet.circuit_components.single_sided_opo_cc.SingleSidedOPO attribute), 131
space (qnet.circuit_components.three_port_kerr_cavity_cc.ThreePortKerrCa attribute), 133
space (qnet.circuit_components.three_port_opo_cc.ThreePortOPO attribute), 135
space (qnet.circuit_components.two_port_kerr_cavity_cc.TwoPortKerrCavi attribute), 136
space (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 137
space() (in module qnet.algebra.operator_algebra), 90
SpaceTooLargeError, 101
SPost (class in qnet.algebra.super_operator_algebra), 109
SPre (class in qnet.algebra.super_operator_algebra), 108
Squeeze (class in qnet.algebra.operator_algebra), 87
srepr() (in module qnet.printing.srepr), 162
SReprPrinter (in module qnet.printing.srepr), 162
start (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
sub_blockstructure (qnet.circuit_components.double_sided_jaynes_cummin attribute), 117
sub_blockstructure (qnet.circuit_components.double_sided_opo_cc.Double attribute), 118
sub_blockstructure (qnet.circuit_components.kerr_cavity_cc.KerrCavity attribute), 120
sub_blockstructure (qnet.circuit_components.linear_cavity_cc.LinearCavity attribute), 124
sub_blockstructure (qnet.circuit_components.relay_cc.Relay attribute), 128
sub_blockstructure (qnet.circuit_components.relay_double_probe_cc.Relay attribute), 129
sub_blockstructure (qnet.circuit_components.single_sided_jaynes_cumming attribute), 130
sub_blockstructure (qnet.circuit_components.three_port_kerr_cavity_cc.Thr attribute), 133
sub_blockstructure (qnet.circuit_components.three_port_opo_cc.ThreePortO attribute), 135
sub_blockstructure (qnet.circuit_components.two_port_kerr_cavity_cc.Two

attribute), 135
sub_blockstructure (qnet.circuit_components.z_probe_cavity_cc.ZProbeCavity attribute), 137
sub_index (qnet.circuit_components.component.SubComponent attribute), 114
SubComponent (class in qnet.circuit_components.component), 114
substitute() (in module qnet.algebra.abstract_algebra), 59
substitute() (qnet.algebra.abstract_algebra.Expression method), 58
substitute_into_symbolic_model_matrices() (in module qnet.misc.kerr_model_matrices), 143
SuperAdjoint (class in qnet.algebra.super_operator_algebra), 108
superadjoint() (qnet.algebra.super_operator_algebra.SuperOperator method), 106
SuperCommutativeHSOrder (class in qnet.algebra.super_operator_algebra), 107
SuperOperator (class in qnet.algebra.super_operator_algebra), 106
SuperOperatorOperation (class in qnet.algebra.super_operator_algebra), 107
SuperOperatorPlus (class in qnet.algebra.super_operator_algebra), 107
SuperOperatorSymbol (class in qnet.algebra.super_operator_algebra), 107
SuperOperatorTimes (class in qnet.algebra.super_operator_algebra), 108
SuperOperatorTimesOperator (class in qnet.algebra.super_operator_algebra), 109
symbolic_heisenberg_eom() (qnet.algebra.circuit_algebra.SLH method), 67
symbolic_liouvillian() (qnet.algebra.circuit_algebra.SLH method), 67
symbolic_master_equation() (qnet.algebra.circuit_algebra.SLH method), 67
SympyCreate() (in module qnet.convert.to_sympy_matrix), 139

T

T (qnet.algebra.matrix_algebra.Matrix attribute), 79
T_a_qp() (in module qnet.misc.kerr_model_matrices), 143
t_ASSIGN (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_COLON (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_COMMA (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_comment() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
t_error() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
t_FCONST (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_FEEDLEFT (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_FEEDRIGHT (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_ICONST (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_ID() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
t_ignore (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_LPAREN (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_NEWLINE() (qnet.qhdl.qhdl_parser.QHDLParser method), 168
T_qp_a() (in module qnet.misc.kerr_model_matrices), 143
t_RPAREN (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
t_SEMI (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
tau (qnet.circuit_components.delay_cc.Delay attribute), 115
temporary_instance_cache() (in module qnet.algebra.abstract_algebra), 61
tensor() (qnet.algebra.hilbert_space_algebra.HilbertSpace method), 75
tensor_decompose_kets() (in module qnet.algebra.state_algebra), 105
tensor_sym (qnet.printing.base.Printer attribute), 160
TensorKet (class in qnet.algebra.state_algebra), 103
term (qnet.algebra.operator_algebra.ScalarTimesOperator attribute), 88
term (qnet.algebra.state_algebra.ScalarTimesKet attribute), 103
term (qnet.algebra.super_operator_algebra.ScalarTimesSuperOperator attribute), 108
tex() (in module qnet.printing.tex), 163
tgrid (qnet.misc.trajectory_data.TrajectoryData attribute), 157
theta (qnet.circuit_components.and_cc.And attribute), 112
theta (qnet.circuit_components.beamsplitter_cc.Beamsplitter attribute), 113
theta (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
theta (qnet.circuit_components.latch_cc.Latch attribute), 121
theta (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125
theta (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
theta_LS0 (qnet.circuit_components.open_lossy_cc.OpenLossy attribute), 125

thetap (qnet.circuit_components.latch_cc.Latch attribute), 121
ThreePortKerrCavity (class in qnet.circuit_components.three_port_kerr_cavity_cc), 133
ThreePortOPO (class in qnet.circuit_components.three_port_opo_cc), 134
tls_space (qnet.circuit_components.double_sided_jaynes_cummings_cc.DoubleSidedJaynesCummings attribute), 117
tls_space (qnet.circuit_components.single_sided_jaynes_cummings_cc.SingleSidedJaynesCummings attribute), 130
to_circuit() (qnet.qhdl.qhdl.Architecture method), 167
to_python() (qnet.qhdl.qhdl.QHDLObject method), 166
to_qhdl() (qnet.qhdl.qhdl.Architecture method), 167
to_qhdl() (qnet.qhdl.qhdl.BasicInterface method), 166
to_qhdl() (qnet.qhdl.qhdl.Component method), 167
to_qhdl() (qnet.qhdl.qhdl.Entity method), 167
to_qhdl() (qnet.qhdl.qhdl.QHDLObject method), 166
to_str() (qnet.misc.trajectory_data.TrajectoryData method), 157
toABCD() (qnet.algebra.circuit_algebra.Circuit method), 66
tokens (qnet.misc.parser.Parser attribute), 145
tokens (qnet.qhdl.qhdl_parser.QHDLParser attribute), 168
toSLH() (qnet.algebra.circuit_algebra.Circuit method), 65
trace() (qnet.algebra.matrix_algebra.Matrix method), 79
TrajectoryData (class in qnet.misc.trajectory_data), 154
TrajectoryParserError, 154
transpose() (qnet.algebra.matrix_algebra.Matrix method), 78
tree() (in module qnet.printing.tree), 163
tree_str() (in module qnet.printing.tree), 163
TrivialKet (in module qnet.algebra.state_algebra), 101
TrivialSpace (in module qnet.algebra.hilbert_space_algebra), 75
try_adiabatic_elimination() (in module qnet.algebra.circuit_algebra), 74
TwoPortKerrCavity (class in qnet.circuit_components.two_port_kerr_cavity_cc), 135

U

UnequalSpaces, 101
unicode() (in module qnet.printing.unicode), 164
unicode_sub_super() (in module qnet.printing.unicode), 164
UnicodePrinter (in module qnet.printing.unicode), 164
update() (qnet.algebra.pattern_matching.MatchDict method), 93
update_registry() (qnet.printing.base.Printer class method), 160

V

vstackm() (in module qnet.algebra.matrix_algebra), 79

W

W (qnet.circuit_components.inverting_fanout_cc.InvertingFanout attribute), 119
W (qnet.circuit_components.mach_zehnder_cc.MachZehnder attribute), 125
W1 (qnet.circuit_components.latch_cc.Latch attribute), 121
W2 (qnet.circuit_components.latch_cc.Latch attribute), 121
W_beta (qnet.circuit_components.pseudo_nand_cc.PseudoNAND attribute), 127
wc() (in module qnet.algebra.pattern_matching), 95
wrap_fqp() (in module qnet.misc.kerr_model_matrices), 143
wrap_Jqp() (in module qnet.misc.kerr_model_matrices), 143
write() (qnet.misc.qsd_codegen.QSDCodeGen method), 149
write() (qnet.misc.trajectory_data.TrajectoryData method), 157
write_component() (in module qnet.circuit_components.library), 123
WrongCDimError, 64
WrongSignatureError, 57

X

X() (in module qnet.algebra.operator_algebra), 89

Y

Y() (in module qnet.algebra.operator_algebra), 89

Z

Z() (in module qnet.algebra.operator_algebra), 90
zero_or_more (qnet.algebra.pattern_matching.Pattern attribute), 95
zero_sym (qnet.printing.base.Printer attribute), 159
ZeroKet (in module qnet.algebra.state_algebra), 101
ZeroOperator (in module qnet.algebra.operator_algebra), 82
zerosm() (in module qnet.algebra.matrix_algebra), 80
ZeroSuperOperator (in module qnet.algebra.super_operator_algebra), 106
ZProbeCavity (class in qnet.circuit_components.z_probe_cavity_cc), 136