qnet Documentation

Release 1.4.1

Author

Jun 02, 2018

Contents

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

1.1 Dependencies

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

0. Python version 2.6 or higher. QNET is still officially a Python 2 package, but migration to Python 3 should not be too hard to achieve.
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.

1.2 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
```

1.3 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

2.1 The Abstract Algebra module

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.

2.1.1 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()*

2.1.2 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*.

2.2 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.

2.2.1 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
>>> 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:

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

```
>>> 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', ''))
```

2.3 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.

2.3.1 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 $\sigma^s_{jk} := |j\rangle_s \langle k|_s$ (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 $X_1 + X_2 + \cdots + X_n$ (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.

2.3.2 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)
```

(continues on next page)

(continued from previous page)

```
    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:

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

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))
```

2.4 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, $\mathbf{L} = (L_1, L_2, \ldots, L_n)^T$ 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 $\mathbf{M}^\dagger = (\mathbf{M}^T)^\sharp = (\mathbf{M}^\sharp)^T$.

2.4.1 Fundamental Circuit Operations

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

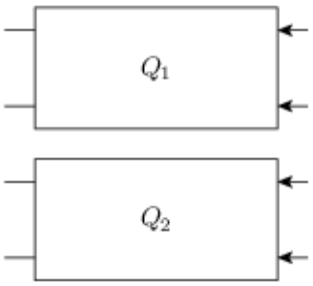


Fig. 1: $Q_1 \boxplus Q_2$

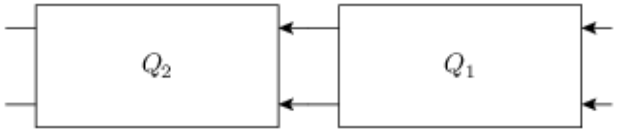


Fig. 2: $Q_2 \lhd Q_1$

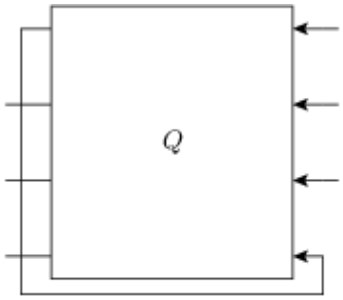


Fig. 3: $[Q]_{1\to 4}$

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.

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:

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

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.

2.4.2 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*.

2.4.3 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
```

(continues on next page)

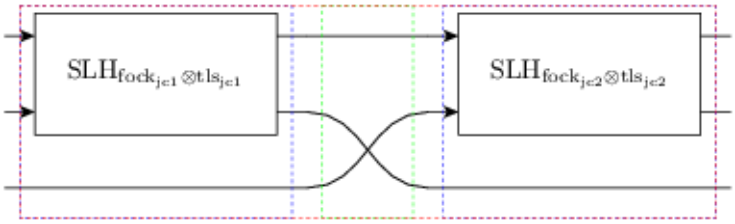
(continued from previous page)

```
        + 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:

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

2.5 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.

2.6 The State (Ket-) Algebra module

This module implements a basic Hilbert space state algebra where by default we represent states ψ as 'Ket' vectors $\psi \rightarrow |\psi\rangle$. 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 $Q = (\mathbf{S}, \mathbf{L}, H)$ its *series inverse* system $Q^{\lhd -1} := (\mathbf{S}^\dagger, -\mathbf{S}^\dagger\mathbf{L}, -H)$

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

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 $Q = (\mathbf{S}, \mathbf{L}, H)$ 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:

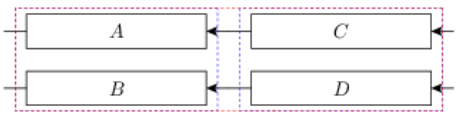


Fig. 1: $(A \boxplus B) \lhd (C \boxplus D)$

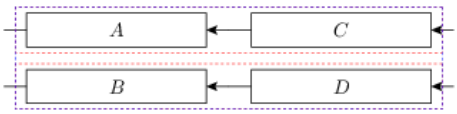


Fig. 2: $(A \lhd C) \boxplus (B \lhd D)$

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 $Q = (\mathbf{S}, \mathbf{L}, H)$ 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 $\{\mathbf{S}_j\}$ and the coupling vectors $\{\mathbf{L}_j\}$ *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).

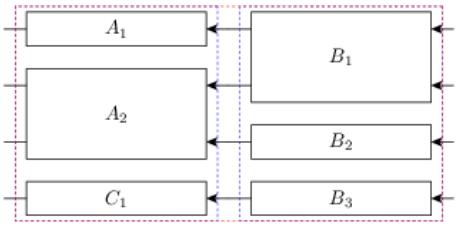


Fig. 3: Series "$(1, 2, 1) \lhd (2, 1, 1)$"

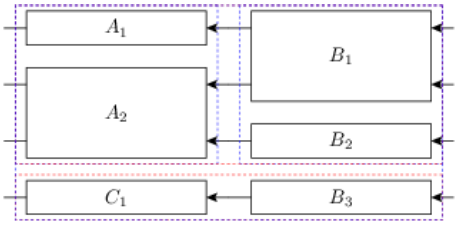


Fig. 4: Optimal decomposition into $(3, 1)$

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.

3.1 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 $\mathbf{P}$, the system described by $(\mathbf{P}, \mathbf{0}, 0)$ represents a pure permutation of the optical fields (ref fig permutation).

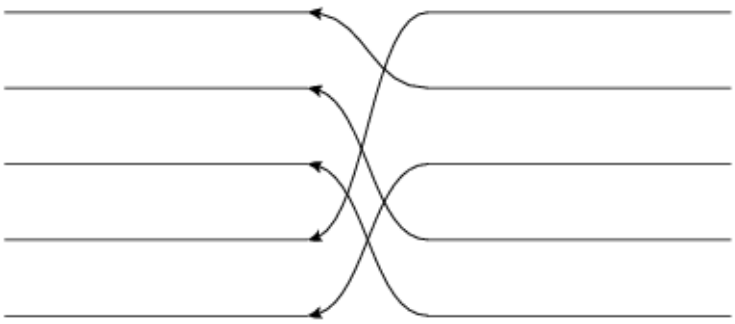


Fig. 5: A graphical representation of $\mathbf{P}_\sigma$ where $\sigma \equiv (4, 1, 5, 2, 3)$ in image tuple notation.

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

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

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 $\mathbf{P}_{\sigma_2 \circ \sigma_1} = \mathbf{P}_{\sigma_2}\mathbf{P}_{\sigma_1}$. This can be shown directly on the order of the matrix elements

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

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 $(\mathbf{P}_\sigma, \mathbf{0}, 0)$. 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 $\mathbf{P}_\sigma^T = \mathbf{P}_{\sigma^{-1}}$ 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 $W_{l\leftarrow k}^{(n)} := P_{\omega_{l\leftarrow k}^{(n)}}$.

3.2 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

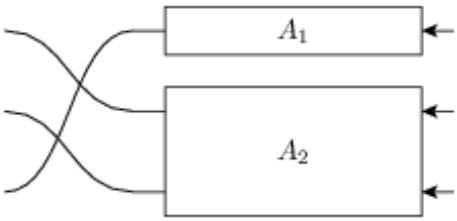


Fig. 6: $P_\sigma \triangleleft (A_1 \boxplus A_2)$

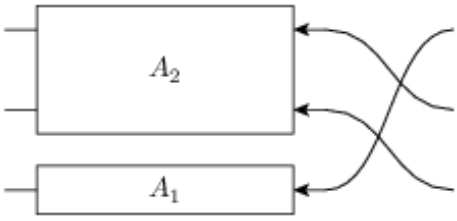


Fig. 7: $(A_2 \boxplus A_1) \triangleleft P_\sigma$

A block permuting series.

Given a block structure $\mathbf{n} := (n_1, n_2, \ldots n_N)$ a permutation $\sigma \in \Sigma_n$ is said to *block permute* $\mathbf{n}$ 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 $\mathbf{n}$ 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 $\sigma = \sigma_\mathrm{b} \circ \sigma_\mathrm{i}$ into a permutation σ_b that block permutes the block structure $\mathbf{n}$ and an internal permutation σ_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

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

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

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

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

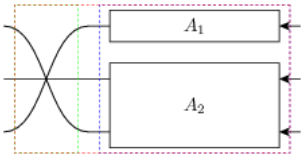


Fig. 8: $P_\sigma \lhd (A_1 \boxplus A_2)$

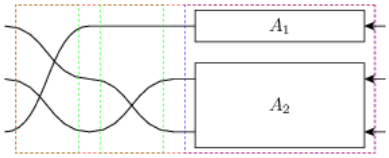


Fig. 9: $P_{\sigma_b} \lhd P_{\sigma_i} \lhd (A_1 \boxplus A_2)$

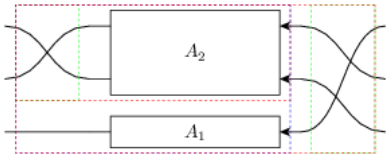


Fig. 10: $((P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_\mathrm{b}}$

A block factorizable series.

A permutation σ block factorizes according to the block structure $\mathbf{n}$ iff for all $1 \leq j \leq N$ we have $\max_{0 \leq k < n_j} \sigma(o_j + k) - \min_{0 \leq k' < n_j} \sigma(o_j + k') = n_j - 1$, 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 $\sigma = \sigma_\mathrm{x} \circ \sigma_\mathrm{b} \circ \sigma_\mathrm{i}$ defining a specific way of constructing both σ_b and σ_i from σ. The remainder σ_x can then be calculated through

$$\sigma_\mathrm{x} := \sigma \circ \sigma_\mathrm{i}^{-1} \circ \sigma_\mathrm{b}^{-1}.$$

Hence, by construction, $\sigma_\mathrm{b} \circ \sigma_\mathrm{i}$ factorizes according to $\mathbf{n}$ 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 σ_b and the decomposable σ_i? We wish to define σ_i such that the remainder $\sigma \circ \sigma_\mathrm{i}^{-1} = \sigma_\mathrm{x} \circ \sigma_\mathrm{b}$ does not cross any two signals that are emitted from the same block. Since by construction σ_b only permutes full blocks anyway this means that σ_x also does not cross any two signals emitted from the same block. This completely determines σ_i and we can therefore calculate $\sigma \circ \sigma_\mathrm{i}^{-1} = \sigma_\mathrm{x} \circ \sigma_\mathrm{b}$ as well. To construct σ_b it is sufficient to define an total order relation on the blocks that only depends on the block structure $\mathbf{n}$ and on $\sigma \circ \sigma_\mathrm{i}^{-1}$. We define the order on the blocks such that they are ordered according to their minimal

image point under σ. Since $\sigma \circ \sigma_{\mathrm{i}}^{-1}$ 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 $\sigma \circ \sigma_{\mathrm{i}}^{-1}(o_j)$. In (ref fig general_factorization) we have illustrated this with an example.

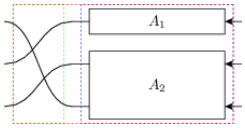


Fig. 11: $P_\sigma \lhd (A_1 \boxplus A_2)$

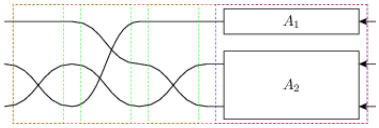


Fig. 12: $P_{\sigma_{\mathrm{x}}} \lhd P_{\sigma_{\mathrm{b}}} \lhd P_{\sigma_{\mathrm{i}}} \lhd (A_1 \boxplus A_2)$

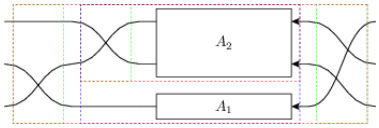


Fig. 13: $(P_{\sigma_{\mathrm{x}}} \lhd (P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_{\mathrm{b}}}$

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

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.

3.3 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.

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.

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 $\mathrm{cdim}\ A = n_A$ and $\mathrm{cdim}\ B = n_B$ and where A and B are not necessarily irreducible. There are four different cases to consider.

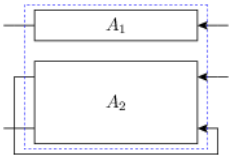


Fig. 14: $[A_1 \boxplus A_2]_{2\to 3}$

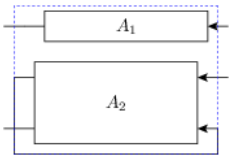


Fig. 15: $A_1 \boxplus [A_2]_{1\to 2}$

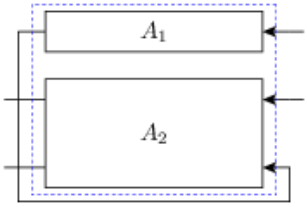


Fig. 16: $[A_1 \boxplus A_2]_{1\to 3}$

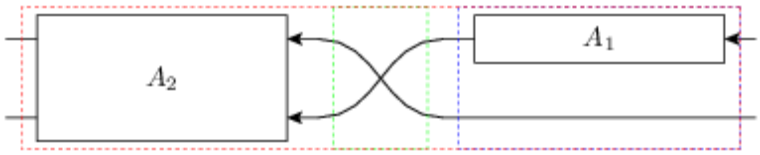


Fig. 17: $A_2 \lhd W^{(2)}_{2\leftarrow 1} \lhd (A_2 \boxplus \mathrm{id}_1)$

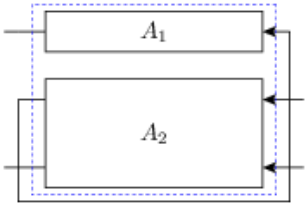


Fig. 18: $[A_1 \boxplus A_2]_{2\to 1}$

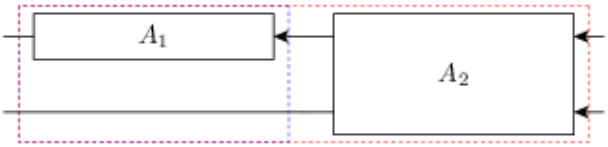


Fig. 19: $(A_1 \boxplus \mathrm{id}_1) \lhd A_2$

- $k, l \leq n_A$: 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 $(\mathrm{id}n_A - 1 \boxplus B) \lhd W^{(n)}_{(l-1)\leftarrow k} \lhd (A + \mathrm{id}n_B - 1)$, cf. Fig. (ref fig fc_re1).
- $l \leq n_A < k$: Again, this corresponds a series but with a reversed order compared to above $(A + \mathrm{id}n_B - 1) \lhd W^{(n)}_{l\leftarrow(k-1)} \lhd (\mathrm{id}n_A - 1 \boxplus B)$, cf. Fig. (ref fig fc_re2).

3.4 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.

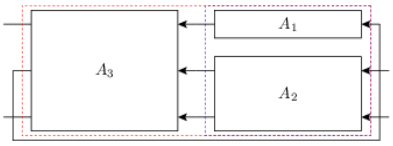


Fig. 20: $[A_3 \lhd (A_1 \boxplus A_2)]_{2\to 1}$

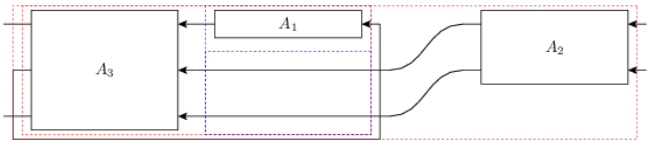


Fig. 21: $(A_3 \lhd (A_1 \boxplus \mathrm{id}_2)) \lhd A_2$

Reduction of series feedback with a concatenation at the RHS

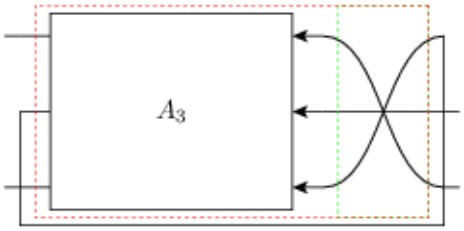


Fig. 22: $[A_3 \lhd P_\sigma]_{2\to 1}$

Reduction of series feedback with a permutation at the RHS

1) $[A \lhd (C \boxplus D)]_{k\to l}$: We define $n_C = \mathrm{cdim}\ C$ and $n_A = \mathrm{cdim}\ A$. Without too much loss of generality, let's assume that $l \leq n_C$ (the other case is quite similar). We can then pull D out of the feedback loop: $[A \lhd (C \boxplus D)]_{k\to l} \longrightarrow [A \lhd (C \boxplus \mathrm{id}n_D)]_{k\to l} \lhd (\mathrm{id}n_C - 1 \boxplus D)$. Obviously, this operation only makes sense if $D \neq \mathrm{id}n_D$. 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 $k \leq n_C$ analogous to above. Provided that $D \neq \mathrm{id}n_D$, we can pull it out of the feedback and get $(\mathrm{id}n_C - 1 \boxplus D) \lhd [(C \boxplus \mathrm{id}n_D) \lhd E]_{k\to l}$.

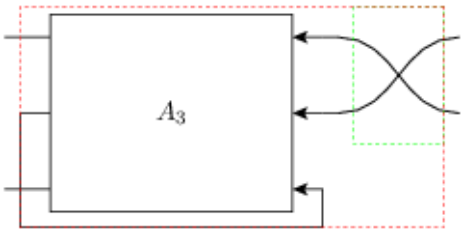


Fig. 23: $[A_3]_{2\to 3} \lhd P_{\tilde{\sigma}}$

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 $[A \lhd W^{(n)}_{j\leftarrow l}]_{k\to l} = [A]_{k\to j}$ which follows from the fact that $W^{(n)}_{j\leftarrow l}$ preserves the order of all incoming signals except the l-th. Now, rewrite

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

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

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

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

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 $\omega^{(n)}_{n\leftarrow k} \circ \sigma \circ \omega^{(n)}_{\sigma^{-1}(k)\leftarrow n}$ 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`

4.1 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:

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

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
```

(continues on next page)

(continued from previous page)

```
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)
```

4.2 Creating custom component symbols for `gschem`

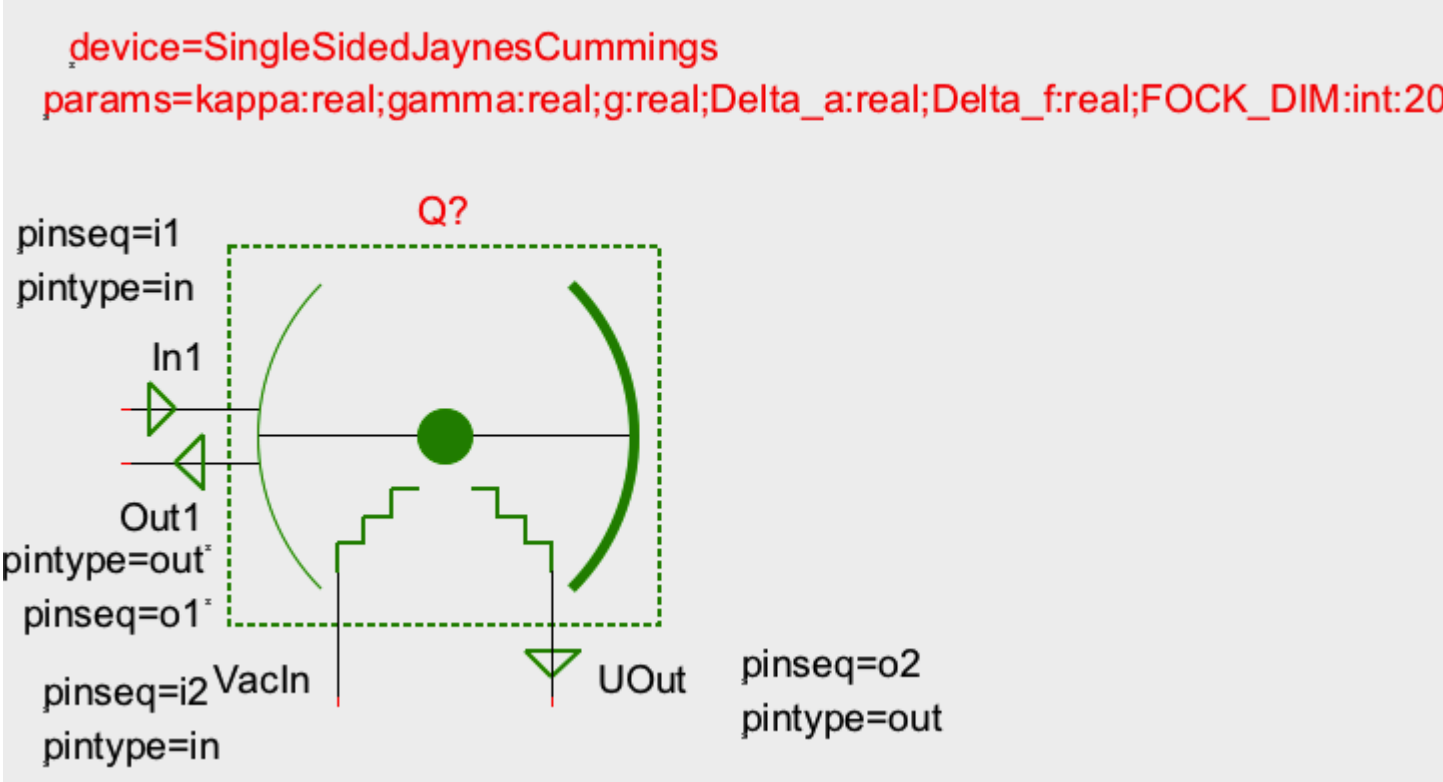
Creating symbols in gschem is similar to the schematic capture process itself:

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

6.1 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.

6.2 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
```

(continues on next page)

(continued from previous page)

```
        [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`.

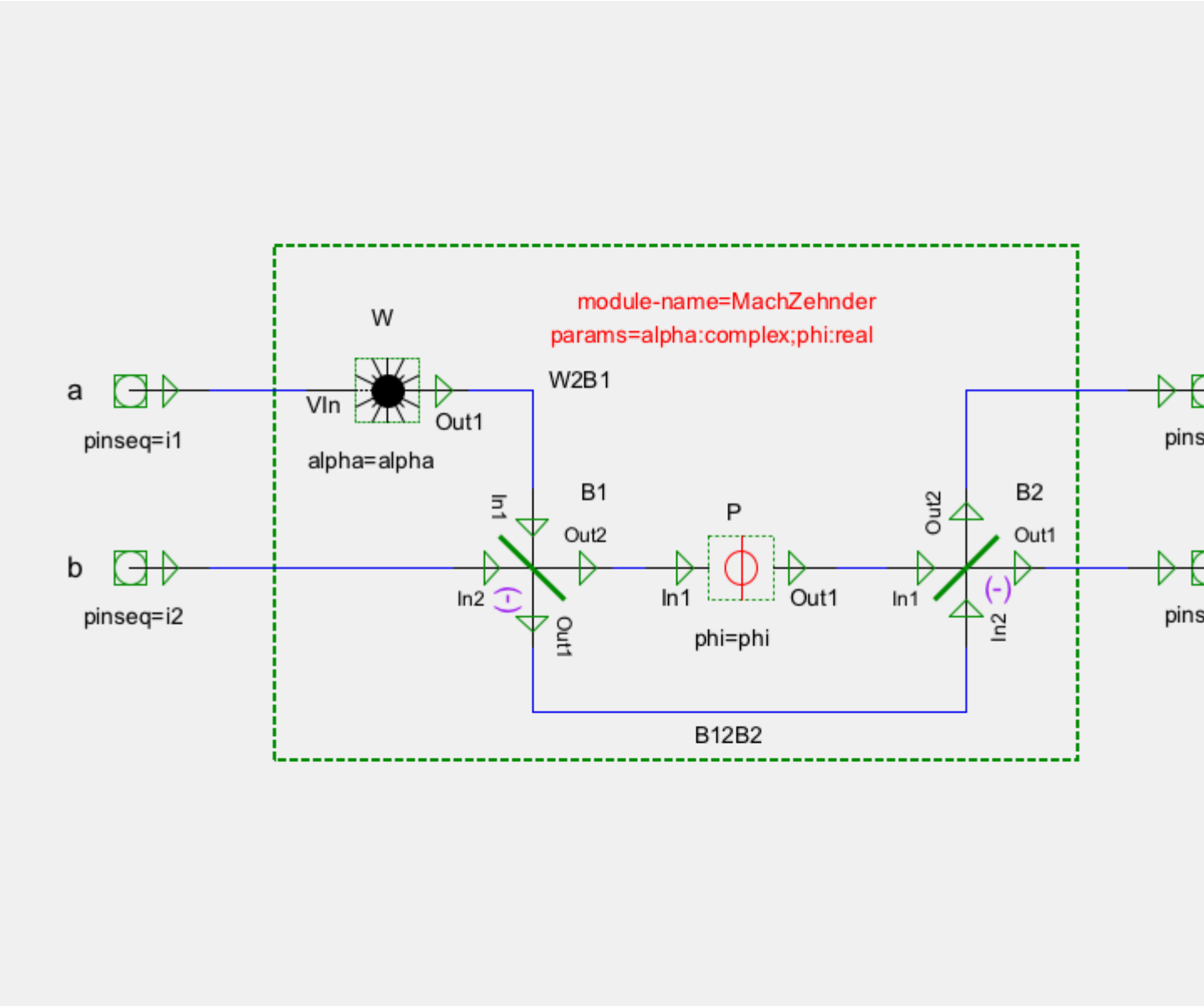
6.2.1 QHDL-Example files:

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
```

(continues on next page)


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

(continued from previous page)

```
    GENERIC (
        alpha : complex);
    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]*.

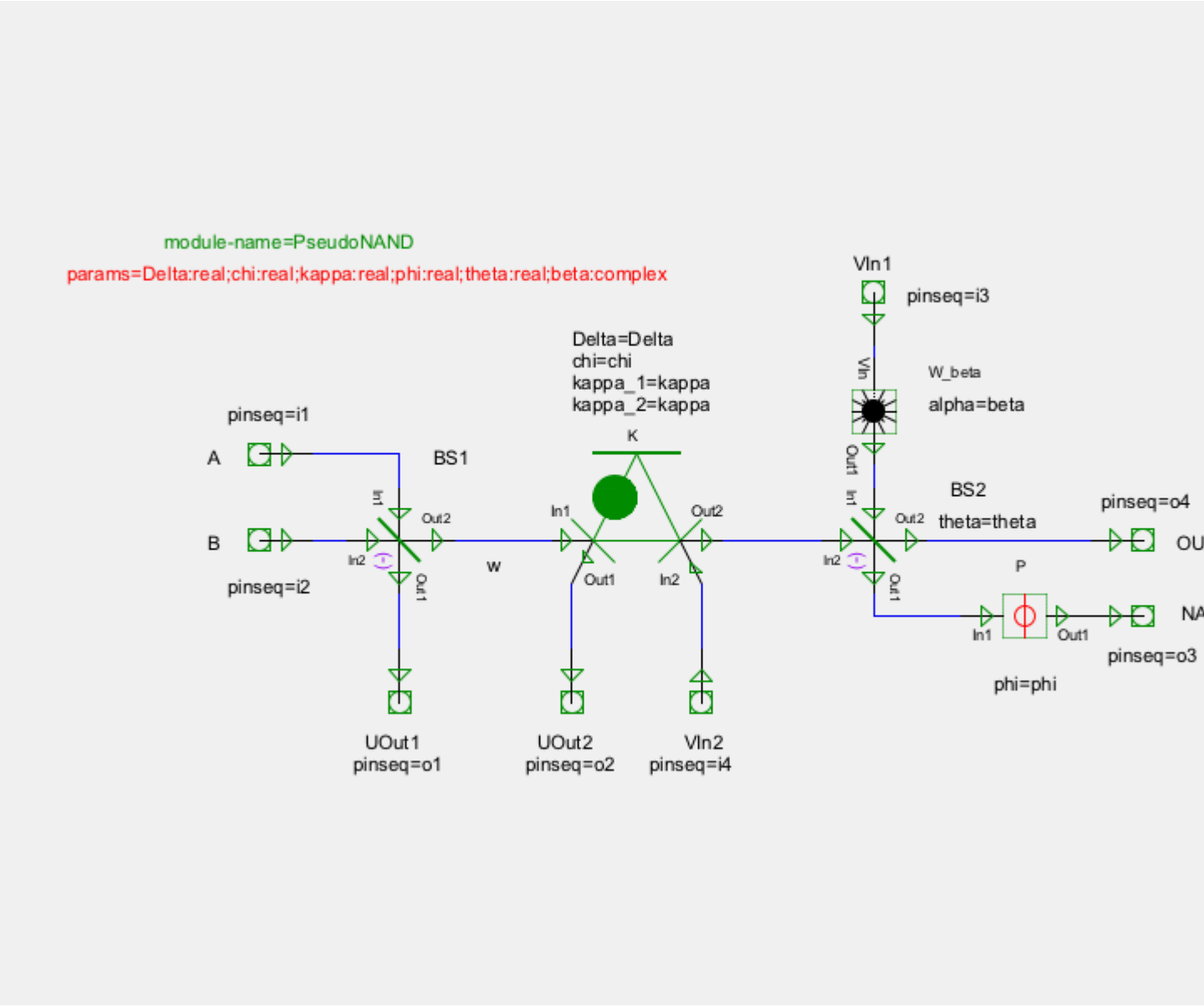


Fig. 1: The `gschem` schematic from which the QHDL file below was automatically created.

```
-- 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
```

(continues on next page)

(continued from previous page)

```
    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);
```

(continues on next page)

(continued from previous page)

```
    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 (
```

(continues on next page)

(continued from previous page)

```
        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

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

8.1.1 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]:

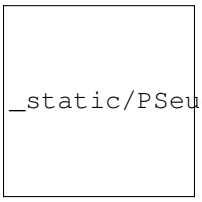
```
N.creduce()
```

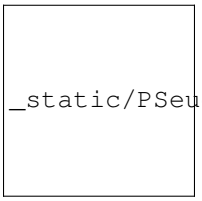
Out[3]:

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

In[4]:

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

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





SLH model

In[5]:

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

Out[5]:

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

Heisenberg equation of motion of the mode operator a

In[6]:

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

Out[6]:

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

In[7]:

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

Out[7]:

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

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]:

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

In[9]:

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

Out[9]:

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

8.1.2 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
```

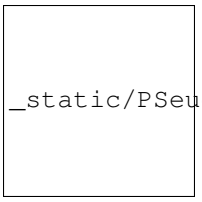
Out[10]:

$$\left[\left(\mathbf{1}_{3}\boxplus\mathrm{N}_{2}\right)\lhd\left(\left(\mathbf{P}_{\sigma}\begin{pmatrix}0&1&2&3\\0&1&3&2\end{pmatrix}\lhd\mathrm{N}_{1}\right)\boxplus\mathbf{1}_{3}\right)\right]_{5\to0}\lhd\left(W(A)\boxplus\mathbf{1}_{2}\boxplus W(B)\boxplus\mathbf{1}_{2}\right)$$

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()
```



SLH model

In[12]:

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

Out[12]:

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

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]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-A-\beta e^{\imath\phi}\cos\left(\theta\right)\right)-\left(\imath\Delta+\kappa\right)a_{\mathrm{N_1.K}}+\frac{1}{2}\sqrt{2}\kappa e^{\imath\phi}\sin\left(\theta\right)a_{\mathrm{N_2.K}}-2\imath\chi a^{\dagger}_{\mathrm{N_1.K}}a_{\mathrm{N_1.K}}a_{\mathrm{N_1.K}}$$

In[16]:

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

Out[16]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(-B-\beta e^{\imath\phi}\cos\left(\theta\right)\right)+\frac{1}{2}\sqrt{2}\kappa e^{\imath\phi}\sin\left(\theta\right)a_{\mathrm{N_1.K}}-\left(\imath\Delta+\kappa\right)a_{\mathrm{N_2.K}}-2\imath\chi a^{\dagger}_{\mathrm{N_2.K}}a_{\mathrm{N_2.K}}a_{\mathrm{N_2.K}}$$

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]:

$$\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(A+\beta e^{\imath\phi}\cos\left(\theta\right)\right)\mathrm{spost}\left[a^{\dagger}_{\mathrm{N_1.K}}\right]+\frac{1}{2}\sqrt{2}\sqrt{\kappa}\left(B+\beta e^{\imath\phi}\cos\left(\theta\right)\right)\mathrm{spost}\left[a^{\dagger}_{\mathrm{N_2.K}}\right]+\frac{\sqrt{2}\sqrt{\kappa}\left(-e^{\imath\phi}\overline{A}-\cos\left(\theta\right)\overline{\beta}\right)}{2e^{\imath\phi}}\mathrm{spost}\left[a_{\mathrm{N_1.K}}\right]$$

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]:

$$\hat{0}$$

8.2 Numerical Analysis via QuTiP

8.2.1 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
```

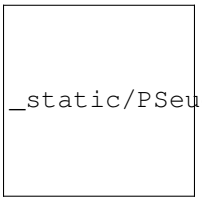
```
100.0%  (1/1)  Est. time remaining: 00:00:00:00
```

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

8.2.2 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]:

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

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))
```

(continues on next page)

(continued from previous page)

```
        # 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]:

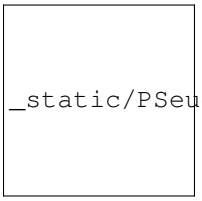
```
datan1 = qutip.expect(observables_qt[0], data)
datan2 = qutip.expect(observables_qt[1], data)
```

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 $", r"$\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

API:

CHAPTER 10

The qnet API

10.1 algebra Package

10.1.1 algebra Package

10.1.2 abstract_algebra Module

Abstract Algebra

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

See *The Abstract Algebra module* for more details.

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.**AlgebraException**
Bases: Exception

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

class qnet.algebra.abstract_algebra.**Expression**
Bases: object

Basic class defining the basic methods any Expression object should implement.

all_symbols()

Returns The set of all_symbols contained within the expression.

Return type set

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

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

tex()
Return a string containing a TeX-representation of self. Note that this needs to be wrapped by '$' characters for 'inline' LaTeX use.

class qnet.algebra.abstract_algebra.**KeyTuple**
Bases: tuple

class qnet.algebra.abstract_algebra.**Match**
Bases: dict

Subclass of dict that overloads the + operator to create a new dictionary combining the entries. It fails when there are duplicate keys.

class qnet.algebra.abstract_algebra.**NamedPattern**(*name*, *pattern*)
Bases: *qnet.algebra.abstract_algebra.Operation*

Create a named (sub-)pattern for later use in processing elements of a matched expression.:

```
NamedPattern(name, pattern)
```

Parameters

- **name** (*str*) – Pattern identifier
- **pattern** (Expression, PatternTuple) – Pattern expression

class qnet.algebra.abstract_algebra.**OperandsTuple**
Bases: tuple

Specialized tuple to store expression operands for the purpose of matching them to patterns.

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

Abstract base class for all operations, where the operands themselves are also expressions.

classmethod create(**operands*)
Instead of directly instantiating an instance of any subclass of Operation, it is advised to call the `create()` classmethod instead. This method takes the same arguments as the constructor, but can pre-process them and even return an object of a different type based on the operands.

Parameters **operands** – The operands for the operation.

operands

Returns The operands of the operation.

Return type tuple

classmethod order_key(*obj*)
Provide a default ordering mechanism for achieving canonical ordering of expressions sequences.

Parameters **obj** – The object to create a key for.

class qnet.algebra.abstract_algebra.**PatternTuple**
Bases: tuple

Specialized tuple to store expression pattern operands.

class qnet.algebra.abstract_algebra.**Wildcard**(*name=''*, *mode=1*, *head=None*, *condition=None*)

Bases: *qnet.algebra.abstract_algebra.Expression*

Basic wildcard expression that can match a single expression or in the context of matching the operands of an Operation object one may match one_or_more or zero_or_more operands with the same wildcards. If the wildcard has a name, a successful match leads to a Match object in which the object that matched the wildcard is stored under that name. One can also restrict the type of the matched Expression by providing a head argument and the condition argument allows for passing a function that performs additional tests on a potential match.

condition = None
extra condition for a successful match (default = None, corresponding to no restriction).

head = None
head/type of the matched object (default = None, corresponding to no restriction).

mode = 1
mode of the wildcard, i.e. how many operands it can match (default = single).

name = ''
name/identifier of the wildcard (default = "").

one_or_more = 2
Value of *Wildcard.mode* for matching one or more operands/objects

single = 1
Value of *Wildcard.mode* for matching single operands/objects

zero_or_more = 3
Value of *Wildcard.mode* for matching zero or more operands/objects

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

Bases: *qnet.algebra.abstract_algebra.AlgebraError*

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

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

Parameters **expr** – The expression to find all_symbols in.

Returns A set of all_symbols within expr.

Return type set

qnet.algebra.abstract_algebra.**assoc**(*dcls*)
Associatively expand out nested arguments of the flat class.

```
>>> @assoc
>>> class Plus(Operation):
>>>     pass
>>> Plus.create(1,Plus(2,3))
    Plus(1,2,3)
```

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._assoc() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**check_signature**(*dcls*)
Check that the operands passed to the create classmethod of an Operation type conform to certain types. For each allowed argument/operand, provide a tuple of types (or one of CLS, DCLS, see extended_isinstance docs). E.g.

```
>>> @check_signature
>>> class X(Operation):
>>>     signature = str, (int, str)
>>>
>>> X.create("1", 2)
    X("1", 2)
>>> X.create("1", "2")
    X("1", "2")
```

The following all raise *WrongSignatureError* exception.

```
>>> X.create("1")
>>> X.create(1, "1")
>>> X.create("1", 2, 3)
```

Automatically generated class decorator based on the method `qnet.algebra.abstract_algebra._check_signature()` using *preprocess_create_with()*.

`qnet.algebra.abstract_algebra.`**`check_signature_assoc`**`(`*dcls*`)`

Like *check_signature()* but for *assoc()*-decorated Operations. In this case the signature need only contain a single entry.

```
>>> @assoc
>>> @check_signature
>>> class X(Operation):
>>>     signature = str
>>> X.create("hello", "you")
    X("hello", "you")
```

The following then raises a *WrongSignatureError* because the third argument is no string

```
>>> X.create("hello", "you", 2)
```

Automatically generated class decorator based on the method `qnet.algebra.abstract_algebra._check_signature_assoc()` using *preprocess_create_with()*.

`qnet.algebra.abstract_algebra.`**`extended_isinstance`**`(`*obj*, *class_info*, *dcls*, *cls*`)`

Like isinstance but with two extra arguments to allow for placeholder objects `(DCLS, CLS)` to stand for the class objects passed as extra arguments `dcls`, `cls`. This allows one to specify a self-referential *signature* class attribute to allow for recursive Operation signatures. E.g.

```
>>> @check_signature
>>> class X(Operation):
>>>     signature = str, X
```

will yield an exception, because X within the class body refers to a class object that has not been defined yet. Instead, one can do

```
>>> @check_signature
>>> class X(Operation):
>>>     signature = str, CLS
```

to refer to the class of the object being instantiated (could be a subclass of `X`), or

```
>>> @check_signature
>>> class X(Operation):
>>>     signature = str, DCLS
```

to always refer to X itself and not a subclass.

Parameters

- **obj** (*object*) – The instance
- **class_info** (*type or tuple of type-objects*) – A type, DCLS, CLS, or a tuple of these
- **dcls** (*type*) – The (super-)class that the signature is defined for.
- **cls** (*type*) – The concrete (sub-)class whose instance is being initialized.

qnet.algebra.abstract_algebra.**filter_neutral**(*dcls*)

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.

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

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._filter_neutral() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**idem**(*dcls*)

Remove duplicate arguments and order them via the cls's order_key key object/function. E.g.

```
>>> @idem
>>> class Set(Operation):
>>>     pass
>>> Set.create(1,2,3,1,3)
    Set(1,2,3)
```

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._idem() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**make_classmethod**(*method*, *cls*)

Make a bound classmethod from an unbound method taking an additional first argument cls

Parameters

- **method** (*FunctionType*) – The unbound method
- **cls** (*type*) – The class to bind it to

Returns Bound class method

Return type MethodType

qnet.algebra.abstract_algebra.**match**(*pattern*, *expr*)

Match a pattern against an expression and return a Match object if successful or False, if not. Works recursively.

Parameters

- **pattern** (Expression *or* PatternTuple) – Pattern expression
- **expr** (Expression *or* OperandsTuple) – Expression to match against the pattern.

Returns Match object or False

Return type *Match* or False

qnet.algebra.abstract_algebra.**match_range**(*pattern*)

Compute how many objects/operands a given pattern can minimally and maximally match.

Parameters **pattern** – The pattern object

Returns min_number, max_number

Return type tuple

Raise ValueError, if unknown pattern mode for Wildcard object

qnet.algebra.abstract_algebra.**match_replace**(*dcls*)

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 wildcards:

```
>>> A = wc("A")
>>> A_float = wc("A", head = float)
```

Then an operation:

```
>>> @match_replace
>>> class Invert(Operation):
>>>     _rules = []
```

Then some _rules:

```
>>> Invert._rules += [
>>>     ((Invert(A),), lambda A: A),
>>>     ((A_float,), lambda A: 1./A),
>>> ]
```

Check rule application:

```
>>> Invert.create("hallo")              # matches no rule
    Invert("hallo")
>>> Invert.create(Invert("hallo"))      # matches first rule
    "hallo"
>>> Invert.create(.2)                   # matches second rule
    5.
```

A pattern can also have the same wildcard appear twice:

```
>>> @match_replace
>>> class X(Operation):
>>>     _rules = [
>>>         ((A, A), lambda A: A),
>>>     ]
```

```
>>> X.create(1,2)
    X(1,2)
```

```
>>> X.create(1,1)
    1
```

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._match_replace() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**match_replace_binary**(*dcls*)

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

```
>>> A = wc("A")
```

```
>>> @match_replace_binary
>>> class FilterDupes(Operation):
>>>     _rules = [
>>>         ((A,A), lambda A: A),
>>>     ]
```

```
>>> FilterDupes.create(1,2,3,4)          # No subsequent duplicates present
    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)        # Some duplicates, but not subsequent
    FilterDupes(1,2,3,2,4)
```

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._match_replace_binary() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**mathematica**(*s*)

qnet.algebra.abstract_algebra.**orderby**(*dcls*)

Re-order arguments via the class's order_key key object/function. Use this for commutative operations: E.g.

```
>>> @orderby
>>> class Times(Operation):
>>>     pass
>>> Times.create(2,1)
    Times(1,2)
```

Automatically generated class decorator based on the method qnet.algebra.abstract_algebra._orderby() using *preprocess_create_with()*.

qnet.algebra.abstract_algebra.**preprocess_create_with**(*method*)

This factory method allows for adding argument pre-processing decorators to a class's create classmethod.

Parameters **method** (*FunctionType*) – A decorating create classmethod f() with signature: f(decorated_class, decorated_method, cls, *args)

Returns A class decorator function that decorates the 'create' classmethod of the decorated class.

Return type FunctionType

qnet.algebra.abstract_algebra.**prod**(*sequence*, *neutral=1*)

Analog of the builtin *sum()* method. :param sequence: Sequence of objects that support being multiplied to each other. :type sequence: Any object that implements __mul__() :param neutral: The initial return value, which is also returned for zero-length sequence arguments. :type neutral: Any object that implements __mul__() :return: The product of the elements of *sequence*

qnet.algebra.abstract_algebra.**set_union**(**sets*)

Similar to sum(), but for sets. Generate the union of an arbitrary number of set arguments.

Parameters **sets** (*set*) – Sets to for the union of.

Returns Union set.

Return type set

qnet.algebra.abstract_algebra.**singleton**(*cls*)
Singleton class decorator. Turns a class object into a unique instance.

Parameters **cls** (*type*) – Class to decorate

Returns The singleton instance of that class

Return type cls

qnet.algebra.abstract_algebra.**substitute**(*expr*, *var_map*)
(Safe) substitute, substitute objects for all symbols.

Parameters

- **expr** – The expression in which to perform the substitution
- **var_map** (*dict*) – The substitution dictionary. See *qnet.algebra.abstract_algebra.substitute()* documentation

qnet.algebra.abstract_algebra.**tex**(*obj*)

Parameters **obj** – Object to represent in LaTeX.

Returns Return a LaTeX string-representation of obj.

Return type str

qnet.algebra.abstract_algebra.**unequals**(*x*)

qnet.algebra.abstract_algebra.**update_pattern**(*expr*, *match_obj*)
Replace all wildcards in the pattern expression with their matched values as specified in a Match object.

Parameters

- **expr** (Expression *or* PatternTuple) – Pattern expression
- **match_obj** (Match) – Match object

Returns Expression with replaced wildcards

Return type *Expression* or *PatternTuple*

qnet.algebra.abstract_algebra.**wc**(*name_mode='_'*, *head=None*, *condition=None*)
Helper function to create a Wildcard object.

Parameters

- **name_mode** (*str*) – Combined name and mode (cf *Wildcard*) argument.
  - `"A"` -> `name="A", mode = Wildcard.single`
  - `"A_"` -> `name="A", mode = Wildcard.single`
  - `"B__"` -> `name="B", mode = Wildcard.one_or_more`
  - `"B___"` -> `name="C", mode = Wildcard.zero_or_more`
- **head** (*tuple or ClassType or None*) – See Wildcard doc
- **condition** (*FunctionType or None*) – See Wildcard doc

Returns A Wildcard object

Return type *Wildcard*

10.1.3 `hilbert_space_algebra` Module

The Hilbert Space Algebra

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*.

exception qnet.algebra.hilbert_space_algebra.**BasisNotSetError**(*local_space*)
Bases: *qnet.algebra.abstract_algebra.AlgebraError*

Is raised when the basis states of a LocalSpace are requested before being defined.

Parameters **local_space** –

class qnet.algebra.hilbert_space_algebra.**HilbertSpace**
Bases: object

Basic Hilbert space class from which concrete classes are derived.

dimension

Returns The full dimension of the Hilbert space

Return type int

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

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

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

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

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

Return type bool

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

local_factors()

Returns A sequence of LocalSpace objects that tensored together yield this Hilbert space.

Return type tuple of LocalSpace objects

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

Parameters **other** (HilbertSpace) – Space to remove

Returns Hilbert space for remaining degrees of freedom.

Return type *HilbertSpace*

tensor(*other*)
Tensor product between Hilbert spaces

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

Returns Tensor product space.

Return type *HilbertSpace*

class qnet.algebra.hilbert_space_algebra.**LocalSpace**(**operands*)

Bases: *qnet.algebra.hilbert_space_algebra.HilbertSpace*, *qnet.algebra.abstract_algebra.Operation*

Basic class to instantiate a local Hilbert space, i.e., for a single degree of freedom.

```
LocalSpace(name, namespace)
```

Parameters

- **name** (*str*) – The identifier of the local space / degree of freedom
- **namespace** (*str*) – The namespace for the degree of freedom, useful in hierarchical system models.

basis

Returns The set of basis states of the local Hilbert space

Return type sequence of int or str

classmethod create(**args*)

Instead of directly instantiating an instance of any subclass of Operation, it is advised to call the `create()` classmethod instead. This method takes the same arguments as the constructor, but can preprocess them and even return an object of a different type based on the operands.

param operands The operands for the operation.

– LocalSpace.create() preprocessed by _check_signature –

Check that the operands passed to the create classmethod of an Operation type conform to certain types. For each allowed argument/operand, provide a tuple of types (or one of `CLS`, `DCLS`, see extended_isinstance docs). E.g.

```
>>> @check_signature
>>> class X(Operation):
>>>     signature = str, (int, str)
>>>
>>> X.create("1", 2)
    X("1", 2)
>>> X.create("1", "2")
    X("1", "2")
```

The following all raise `WrongSignatureError` exception.

```
>>> X.create("1")
>>> X.create(1, "1")
>>> X.create("1", 2, 3)
```

dimension

The local state space dimension.

signature = (<class 'str'>, <class 'str'>)

class qnet.algebra.hilbert_space_algebra.**ProductSpace**(**operands*)

Bases: *qnet.algebra.hilbert_space_algebra.HilbertSpace*, *qnet.algebra.abstract_algebra.Operation*

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

```
ProductSpace(*factor_spaces)
```

Parameters **factor_spaces** (HilbertSpace) – The Hilbert spaces to be tensored together.

classmethod create(**args*)

None – ProductSpace.create() preprocessed by _filter_neutral –

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.

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

– ProductSpace.create() preprocessed by _check_signature_assoc –

Like `check_signature()` but for `assoc()`-decorated Operations. In this case the signature need only contain a single entry.

```
>>> @assoc
>>> @check_signature
>>> class X(Operation):
>>>     signature = str
>>> X.create("hello", "you")
    X("hello", "you")
```

The following then raises a `WrongSignatureError` because the third argument is no string

```
>>> X.create("hello", "you", 2)
```

– ProductSpace.create() preprocessed by _idem –

Remove duplicate arguments and order them via the cls's order_key key object/function. E.g.

```
>>> @idem
>>> class Set(Operation):
>>>     pass
>>> Set.create(1,2,3,1,3)
    Set(1,2,3)
```

– ProductSpace.create() preprocessed by convert_to_spaces_mtd –

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.

– ProductSpace.create() preprocessed by _assoc –

Associatively expand out nested arguments of the flat class.

```
>>> @assoc
>>> class Plus(Operation):
>>>     pass
>>> Plus.create(1,Plus(2,3))
    Plus(1,2,3)
```

neutral_element = TrivialSpace

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

qnet.algebra.hilbert_space_algebra.**convert_to_spaces_mtd**(*dcls*, *clsmtd*, *cls*, **ops*)
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.**local_space**(*name*, *namespace=''*, *dimension=None*, *basis=None*)
Create a LocalSpace with by default empty namespace string. If one also provides a set of basis states, these get stored via the BasisRegistry object. ALternatively, one may provide a dimension such that the states are simply labeled by a range of integers:

```
[0, 1, 2, ..., dimension -1]
```

Parameters

- **name** (*str or int*) – Local space identifier
- **namespace** (*str*) – Local space namespace, see LocalSpace documentation
- **dimension** (*int*) – Dimension of local space (optional)
- **basis** (*sequence of int or sequence of str*) – Basis state labels for local space

10.1.4 operator_algebra Module

10.1.5 permutations Module

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.**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.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.**compose_permutations**(*alpha*, *beta*)

Find the composite permutation

$$\begin{aligned} \sigma &:= \alpha \cdot \beta \\ \Leftrightarrow \sigma(j) &= \alpha\left(\beta(j)\right) \end{aligned}$$

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.**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.**invert_permutation**(*permutation*)

Compute the image tuple of the inverse permutation. :param permutation: A valid (cf. :py:func:check_permutation) permutation. :return: The inverse permutation tuple :rtype: tuple

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.**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_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.**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

10.1.6 circuit_algebra Module

10.1.7 state_algebra Module

10.1.8 super_operator_algebra Module

10.2 circuit_components Package

10.2.1 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.

10.2.2 beamsplitter_cc Module

10.2.3 component Module

10.2.4 delay_cc Module

10.2.5 displace_cc Module

10.2.6 double_sided_opo_cc Module

10.2.7 kerr_cavity_cc Module

10.2.8 library Module

This module features some helper functions for automatically creating and managing a library of circuit component definition files.

`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.**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.**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

10.2.9 linear_cavity_cc Module

10.2.10 mach_zehnder_cc Module

10.2.11 open_lossy_cc Module

10.2.12 phase_cc Module

10.2.13 pseudo_nand_cc Module

10.2.14 pseudo_nand_latch_cc Module

10.2.15 relay_cc Module

10.2.16 single_sided_jaynes_cummings_cc Module

10.2.17 single_sided_opo_cc Module

10.2.18 three_port_opo_cc Module

10.2.19 zprobe_cavity_cc Module

10.3 misc Package

10.3.1 misc Package

10.3.2 circuit_visualization Module

10.3.3 parse_circuit_strings Module

10.3.4 parser Module

Generic Parser class.

class qnet.misc.parser.**Parser**(***kw*)
Bases: object

Base class for a lexer/parser that has the _rules defined as methods

parse(*inputstring*)

parse_file(*filename*)

precedence = ()

tokens = ()

exception qnet.misc.parser.**ParsingError**
Bases: SyntaxError

Raised for parsing error.

10.3.5 qsd_codegen Module

10.3.6 trajectory_data Module

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 reproducability.

Attribute ID A unique ID for the current state of the TrajectoryData (read-only). See property documentation below

Attribute table A table (OrderedDict of column names to numpy arrays) 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.

Attribute dt Time step between data points

Attribute nt Number of time steps / data points

Attribute operators

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)]"

Attribute record A copy of the complete record of how the averaged expectation values for all operators were obtained. See indepth discussion in the property documentation below.

Attribute col_width 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*

ID
A unique RFC 4122 complient 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

col_width = 25

copy()
Return a (deep) copy of the current object

dt
Time step between data points

extend(*other*)
Extend data with another Trajectory data set, averaging the expectation values. Equivalently to `traj1.extend(traj2)`, the syntax `traj1 += traj2` may be used.

Raises **ValueError** – if data in self and other are incompatible

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(str) => str*) – dictionary (preferrably 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** – directory to which the filenames in *operators* are relative to

'type workdir: str

Raises **ValueError** – if any of the datafiles 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 indentical 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.

n_trajectories(*operator*)
Return the total number of trajectories for the given operator

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.

nt
Number of time steps / data points

operators
Iterator over all operators

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

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 used the `QSDCodeGen` class to set up a QSD propagation. Two observables 'X1', 'X2', have been set up to be written to file 'X1.out', and 'X2.out'. The *QSDCodeGen.set_trajectories* method has been called with *n_trajectories=10*, after which a call to *QSDCodeGen.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* (in addition to the existing observables X1, X2), and run the *QSDCodeGen.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.

record_IDs
Set of all IDs in the record

record_seeds
Set of all random number generator seeds in the record

shape
"Tuple (n_row, n_cols) for the data in self.table. The time grid is included in the column count

tgrid
Time grid, as numpy array

to_str(*show_rows=-1*)
Generate full string represenation of 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

exception qnet.misc.trajectory_data.**TrajectoryParserError**
Bases: `Exception`

Exception raised if a TrajectoryData file is malformed

10.4 qhdl Package

10.4.1 qhdl Package

10.4.2 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*.

class qnet.qhdl.qhdl.**Architecture**(*identifier*, *entity*, *components*, *signals*, *assignments*, *global_assignments={}*)

Bases: *qnet.qhdl.qhdl.QHDLObject*

global_in = {}

global_inout = {}

global_out = {}

in_to_signal = {}

inout_to_signal = {}

lossy_signals = []

out_to_signal = {}

signal_to_global_in = {}

signal_to_global_out = {}

signals = []

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*)

class qnet.qhdl.qhdl.**BasicInterface**(*identifier*, *generics*, *ports*)

Bases: *qnet.qhdl.qhdl.QHDLObject*

cid = 0

generic_identifiers

The generic_identifiers property.

generics_to_qhdl(*tab_level*)

gids

The generic_identifiers property.

in_port_identifiers = []

inout_port_identifiers = []

out_port_identifiers = []

port_identifiers

The port_identifiers property.

ports_to_qhdl(*tab_level*)

to_qhdl(*tab_level*)

class qnet.qhdl.qhdl.**Component**(*identifier*, *generics*, *ports*)
Bases: *qnet.qhdl.qhdl.BasicInterface*

to_qhdl(*tab_level=0*)

class qnet.qhdl.qhdl.**Entity**(*identifier*, *generics*, *ports*)
Bases: *qnet.qhdl.qhdl.BasicInterface*

to_qhdl(*tab_level=0*)

exception qnet.qhdl.qhdl.**QHDLError**
Bases: Exception

class qnet.qhdl.qhdl.**QHDLObject**
Bases: object

to_python()

to_qhdl()

qnet.qhdl.qhdl.**dict_keys_sorted_by_val**(*dd*)

qnet.qhdl.qhdl.**gtype_compatible**(*c_t*, *g_t*)

qnet.qhdl.qhdl.**my_debug**(*msg*)

10.4.3 qhdl_parser Module

The PLY-based QHDLParser class.

class qnet.qhdl.qhdl_parser.**QHDLParser**(***kw*)
Bases: *qnet.misc.parser.Parser*

create_circuit_lib(*arch_id=None*)

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_complex(*p*)
complex : LPAREN simple_number COMMA simple_number RPAREN

p_component_declaration(*p*)
component_declaration : COMPONENT ID generic_clause port_clause END COMPONENT opt_id SEMI

p_component_declaration_list(*p*)

component_declaration_list [component_declaration_list component_declaration]

component_declaration

p_empty(*p*)
empty :

p_entity_declaration(*p*)
entity_declaration : ENTITY ID IS generic_clause port_clause END opt_entity opt_id SEMI

p_error(*p*)

p_feedleft_assignment(*p*)

feedleft_assignment [ID FEEDLEFT ID SEMI]

empty

p_feedleft_assignment_list(*p*)

feedleft_assignment_list [feedleft_assignment_list feedleft_assignment]

feedleft_assignment

p_feedright_generic_assignment(*p*)

feedright_generic_assignment [ID FEEDRIGHT id_or_value]

id_or_value

p_feedright_generic_assignment_list(*p*)

feedright_generic_assignment_list [feedright_generic_assignment_list COMMA feedright_generic_assignment]

feedright_generic_assignment

p_feedright_port_assignment(*p*)

feedright_port_assignment [ID FEEDRIGHT ID]

ID

p_feedright_port_assignment_list(*p*)

feedright_port_assignment_list [feedright_port_assignment_list COMMA feedright_port_assignment]

feedright_port_assignment

p_generic_clause(*p*)

generic_clause [generic_statement]

empty

p_generic_default(*p*)

generic_default [ASSIGN number]

empty

p_generic_entry_group(*p*)
generic_entry_group : id_list COLON generic_type generic_default

p_generic_list(*p*)

generic_list [generic_list SEMI generic_entry_group]

generic_entry_group

p_generic_map(*p*)

generic_map [GENERIC MAP LPAREN feedright_generic_assignment_list RPAREN SEMI]

empty

p_generic_statement(*p*)
generic_statement : GENERIC LPAREN generic_list opt_semi RPAREN SEMI

p_generic_type(*p*)
generic_type : REAL | COMPLEX | INT

p_id_list(*p*)

id_list [id_list COMMA ID]

ID

p_id_or_value(*p*)

id_or_value [ID]

number

p_instance_mapping_assignment(*p*)
instance_mapping_assignment : ID COLON ID generic_map port_map

p_instance_mapping_assignment_list(*p*)

instance_mapping_assignment_list [instance_mapping_assignment_list instance_mapping_assignment]

instance_mapping_assignment

p_int(*p*)
int : ICONST

p_io_port_entry_group(*p*)
io_port_entry_group : id_list COLON INOUT signal_type

p_non_io_port_entry_group(*p*)
non_io_port_entry_group : id_list COLON signal_direction signal_type

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_number(*p*)

number [simple_number]

complex

p_opt_arch(*p*)

opt_arch [ARCHITECTURE]

empty

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_port_clause(*p*)

port_clause [port_statement]

empty

p_port_list(*p*)

port_list [with_io_port_list]

non_io_port_list

p_port_map(*p*)

port_map [PORT MAP LPAREN feedright_port_assignment_list RPAREN SEMI]

empty

p_port_statement(*p*)
port_statement : PORT LPAREN port_list opt_semi RPAREN SEMI

p_real(*p*)
real : FCONST

p_signal_direction(*p*)

signal_direction [IN]

OUT

p_signal_entry_group(*p*)
signal_entry_group : SIGNAL id_list COLON signal_type SEMI

p_signal_list(*p*)

signal_list [signal_list signal_entry_group]

signal_entry_group

p_signal_type(*p*)

signal_type [FIELDMODE]

LOSSY_FIELDMODE

p_simple_number(*p*)

simple_number [int]

real

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_with_io_port_list(*p*)

with_io_port_list [io_port_entry_group SEMI non_io_port_list]

io_port_entry_group

parse(*inputstring*)

```
reserved = {'architecture':  'ARCHITECTURE', 'begin':  'BEGIN', 'complex':  'COMPLEX',
start = 'top_level_list'
t_ASSIGN = ':='
t_COLON = ':'
t_COMMA = ','
t_FCONST = '-?((\\d+)(\\.\\d+)(e(\\+|-)?(\\d+))? | (\\d+)e(\\+|-)?(\\d+))'
t_FEEDLEFT = '<='
t_FEEDRIGHT = '=>'
t_ICONST = '-?\\d+'
```

t_ID(*t*)
[_A-Za-z][w_]*

```
t_LPAREN = '\\('
```

t_NEWLINE(*t*)
n+

```
t_RPAREN = '\\)'
t_SEMI = ';'
```

t_comment(*t*)
–[^n]*

t_error(*t*)

```
t_ignore = ' \t\x0c'
tokens = ['INT', 'MAP', 'BEGIN', 'GENERIC', 'OUT', 'OF', 'LOSSY_FIELDMODE', 'COMPONENT
```

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

Index

Q

R

S

T

U

W

Z