QNET

Release 2.0.0-dev

Dec 06, 2018

Contents:

CHAPTER 1

QNET

Computer algebra package for quantum mechanics and photonic quantum networks

Development of QNET happens on Github. You can read the full documentation at ReadTheDocs.

1.1 Features

- Extensible computer algebra system for quantum operators, quantum states, super operators
- Building on SymPy for scalar symbolic algebra
- Implementation of Gough and James' SLH algebra for photonic quantum circuits
- Designed for use within the Jupyter notebook
- Publication-ready, configurable rendering of mathematical formulas
- Conversion to QuTiP objects for numerical simulation

Note that version 2.0 of QNET is a major redesign. See *History* for details.

1.2 Dependencies

- Python version 3.5 or higher. The last version of QNET to support Python 2 is 1.4.3.
- 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.
- The NumPy package for numerical calculations
- Optional: 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 Operator expressions where all symbolic scalar parameters have been replaced by numeric ones, can be converted to (sparse) numeric matrix representations, which are then used to solve for the system dynamics using the tools provided by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.

- Optional: The PyX python package for visualizing circuit expressions as box/flow diagrams. This requires a LaTeX installation on your system. On Linux/Macos and Windows TeX Live and MiKTeX are recommended, respectively.

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

1.3 Installation

To install the latest released version of QNET, run this command in your terminal:

```
$ pip install qnet
```

This is the preferred method to install QNET, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

To install the latest development version of QNET from Github.

```
$ pip install git+https://github.com/mabuchilab/qnet.git@develop#egg=qnet
```

1.4 Usage

To use QNET in a project:

```
import qnet
```

CHAPTER 2

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

2.1 Types of Contributions

2.1.1 Report Bugs

Report bugs at https://github.com/mabuchilab/QNET/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

2.1.2 Fix Bugs / Implement Features

Look through the GitHub issues for bugs or feature requests. Anybody is welcome to submit a pull request for open issues.

2.1.3 Write Documentation

QNET could always use more documentation, whether as part of the official QNET docs, in docstrings, or even on the web in blog posts, articles, and such.

2.1.4 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mabuchilab/QNET/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.2 Get Started!

Ready to contribute? Follow Aaron Meurer's Git Workflow Notes (with `mabuchilab/QNET` instead of `sympy/sympy`)

In short,

1. Clone the repository from `git@github.com:mabuchilab/QNET.git`
2. Fork the repo on GitHub to your personal account.
3. Add your fork as a remote.
4. Pull in the latest changes from the develop branch.
5. Create a topic branch
6. Make your changes and commit them (testing locally)
7. Push changes to the topic branch on *your* remote
8. Make a pull request against the base develop branch through the Github website of your fork.

The project contains a `Makefile` to help with development tasts. In your checked-out clone, do

```
$ make help
```

to see the available make targets.

It is strongly recommended that you use the conda package manager. The `Makefile` relies on conda to create local testing and documentation building environements (`make test` and `make docs`).

Alternatively, you may use `make develop-test` and `make develop-docs` to run the tests or generate the documentation within your active Python environment. You will have to ensure that all the necessary dependencies are installed. Also, you will not be able to test the package against all supported Python versions. You still can (and should) look at https://travis-ci.org/mabuchilab/QNET/ to check that your commits pass all tests.

2.3 Branching Model

QNET uses the git-flow branching model. That is, the `develop` branch takes the role of `master` in the Git Workflow Notes.

In order to create topic branches with `git flow`, after cloning the `qnet` repository, you should initialize it as follows:

```
$ git checkout master
$ git flow init
$ git checkout develop
```

2.4 Testing

QDYN's uses pytest for testing. The test-suite for all supported Python versions is run with

```
$ make test
```

This creates a conda environment for each supported Python version in `./.venv`, installs the QDYN package and all prerequisites into that environment, and runs `py.test`.

In order run a specific test, you may invoke `py.test` manually with the appropriate options, e.g.

```
$ ./.venv/py36/bin/py.test -s -x ./tests/algebra/test_abstract_algebra.py
```

2.5 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. Check https://travis-ci.org/mabuchilab/QNET/pull_requests and make sure that the tests pass for all supported Python versions.

CHAPTER 3

Credits

Hideo Mabuchi had the initial idea for a software package that could exploit the Gough-James SLH formalism to generate an overall open quantum system model for a quantum feedback network based solely on its topology and the component models in analytic form. The actual QNET package was then planned and implemented by Nikolas Tezak. In the Fall of 2015 Michael Goerz joined as a main developer.

Work on QNET was directly supported by DARPA-MTO under Award No. N66001-11-1-4106. Nikolas Tezak was also supported by a Simons Foundation Math+X fellowship as well as a Stanford Graduate Fellowship. Michael Goerz was supported in part by ASD(R&E) under their Quantum Science and Engineering Program (QSEP), and by the Army High Performance Computing Research Center (AHPCRC) (sponsored by the U.S. Army Research Laboratory under contract No. W911NF-07-2-0027). Currently, Michael Goerz is sponsored by the Army Research Laboratory under Cooperative Agreement Number W911NF-16-2-0147.

3.1 Development Lead

- Nikolas Tezak <nikolas@rigetti.com>
- Michael Goerz <mail@michaelgoerz.net>

3.2 Contributors

The following people contributed to to the development of QNET, conceptually, through bug reports, or with code commits.

- Michael Armen
- Armand Niederberger
- Joe Kerckhoff
- Dmitri Pavlichin
- Gopal Sarma

- Ryan Hamerly
- Michael Hush
- Anubhab Haldar
- Gil Tabak
- Edwin Ng
- Tatsuhiro Onodera
- Daniel Wennberg

CHAPTER 4

History

The original 1.0 relase of QNET centered around an implementation of the Quantum Hardware Description Language (QHDL) that serves to describe a circuit topology and specification of a larger entity in terms of parametrizable subcomponents. This is strongly analogous to the specification of electric circuitry using the structural description elements of VHDL or Verilog.

Version 2.0 of QNET shifts the focus of the package to provide a broad symbolic algebra package for quantum mechanics, and the implementation of the SLH circuit algebra. Support of QHDL was removed from QNET, with the intention of re-implementing it in a separate QHDL package, that works on top of QNET. The split was made because the two aspects of the original QNET package serves two different audiences: The basic algebraic tools are will be used by theorists or for numerical models, while QHDL, the definition of circuit components, or the use of the gEDA `gschem` tool are primarily of interest for experimentalists. By developing these two aspects in different packages, we hope the better address the particular needs of each user group.

If you are currently using QHDL through QNET 1.0, you should not upgrade to QNET 2.0. Also, QNET 2.0 drops support for Python 2.

QNET uses Semantic Versioning.

4.1 1.0.0

- initial release

4.2 2.0.0

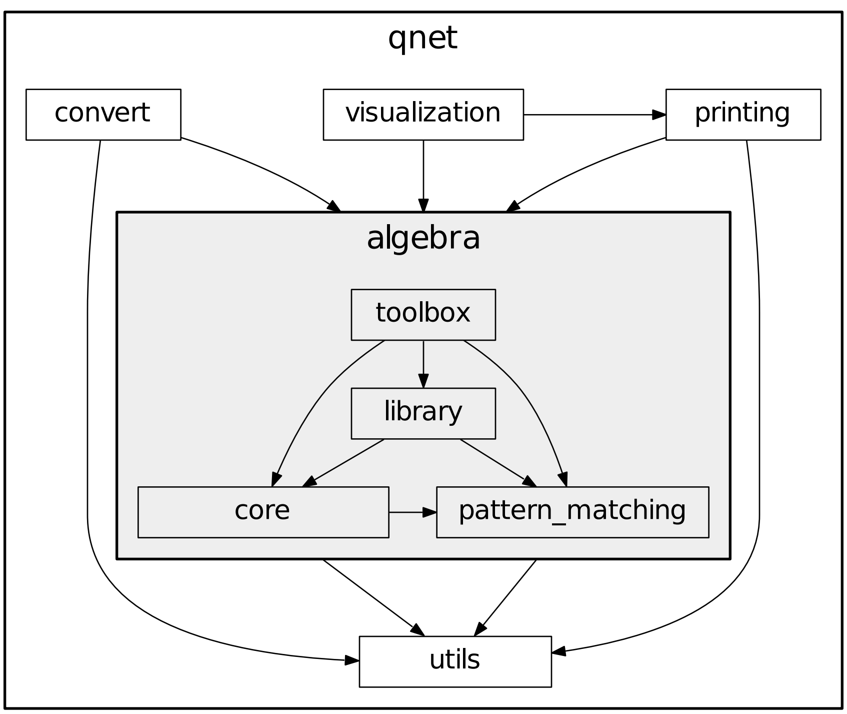
- major restructuring
- drop Python 2 support
- remove support for parsing the quantum-hardware-description-language (QHDL) and the circuit component library. QNET now provides only the fundamental algebraic tools. The QHDL functionality will be extended in a separate future QHDL package

- a new printing system

CHAPTER 5

Library Structure

5.1 Subpackage Organization



QNET is organized into the sub-packages outlined in the above diagram. Each package may in turn contain several sub-modules. The arrows indicate which package imports from which other package.

Every package exports all public symbol from all of its sub-packages/-modules in a "flat" API. Thus, a user can directly import from the top-level *qnet* package.

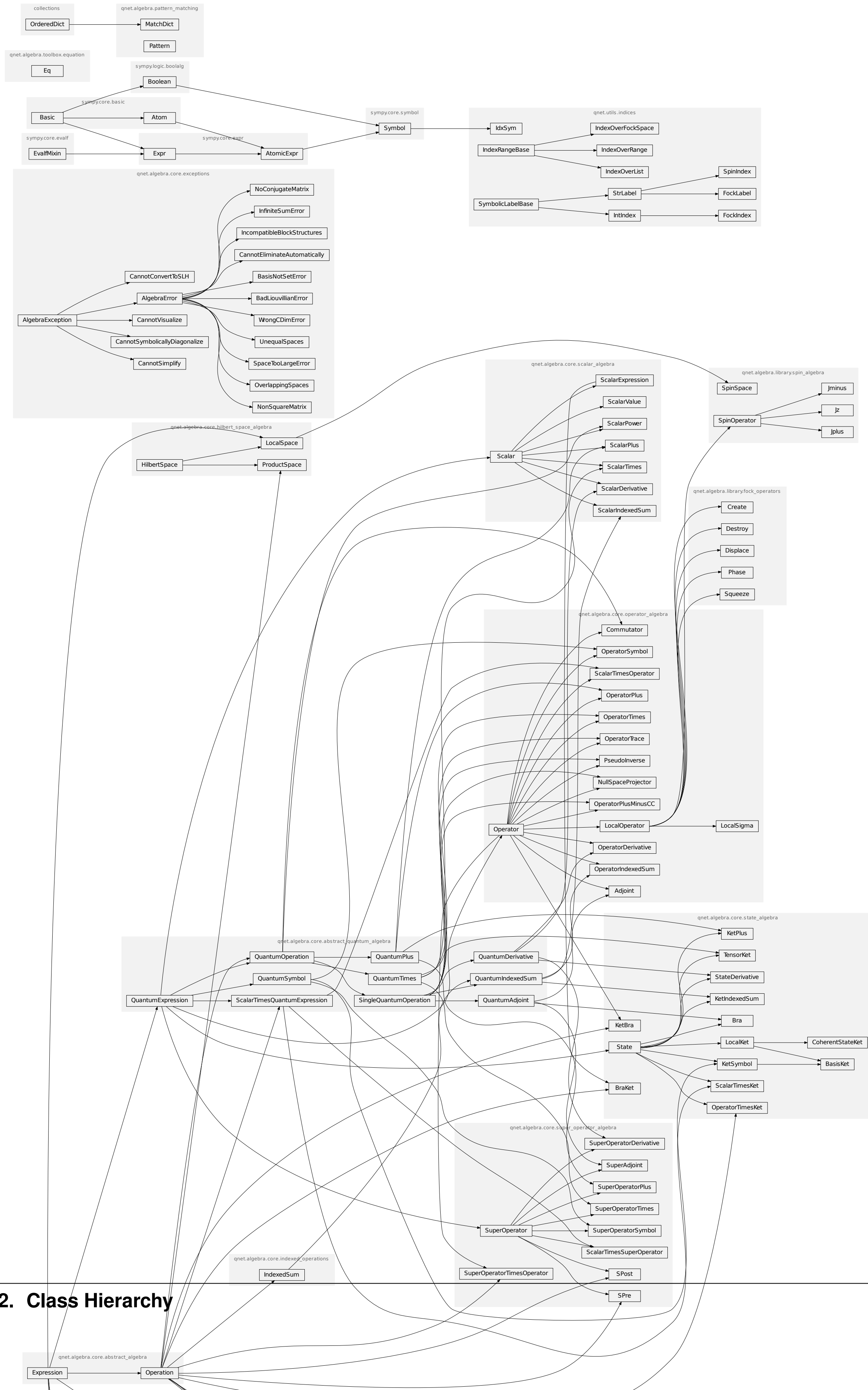
In order from high-level to low-level:

`qnet`	Main QNET package
`qnet.convert`	Conversion to QuTiP and Sympy
`qnet.visualization`	Visualization routines, e.g.
`qnet.printing`	Printing system for QNET Expressions and related objects
`qnet.algebra`	Symbolic quantum and photonic circuit (SLH) algebra
`qnet.algebra.toolbox`	Collection of tools to manually manipulate algebraic expressions
`qnet.algebra.library`	Collection of algebraic objects extending `core`
`qnet.algebra.core`	The fundamental object hiearchies that constitute QNET's various algebras
`qnet.algebra.pattern_matching`	QNET's pattern matching engine.
`qnet.utils`	Auxiliary utilities, mostly for internal use

See also the full modindex

5.2 Class Hierarchy

The following is an inheritance diagram of *all* the classes defined in QNET (this is best viewed as the full-page SVG):


collections
OrderedDict
qnet.algebra.pattern_matching
MatchDict
Pattern
qnet.algebra.toolbox.equation
Eq
sympy.logic.boolalg
Boolean
sympy.core.basic
Basic
Atom
sympy.core.symbol
Symbol
qnet.utils.indices
IdxSym
IndexOverFockSpace
IndexRangeBase
IndexOverRange
IndexOverList
SpinIndex
SymbolicLabelBase
StrLabel
FockLabel
IntIndex
FockIndex
sympy.core.evalf
EvalfMixin
sympy.core.expr
Expr
AtomicExpr
qnet.algebra.core.exceptions
NoConjugateMatrix
InfiniteSumError
IncompatibleBlockStructures
CannotEliminateAutomatically
CannotConvertToSLH
BasisNotSetError
AlgebraError
BadLiouvillianError
AlgebraException
CannotVisualize
WrongCDimError
CannotSymbolicallyDiagonalize
UnequalSpaces
CannotSimplify
SpaceTooLargeError
OverlappingSpaces
NonSquareMatrix
qnet.algebra.core.scalar_algebra
ScalarExpression
ScalarValue
ScalarPower
ScalarPlus
Scalar
ScalarTimes
ScalarDerivative
ScalarIndexedSum
qnet.algebra.library.spin_algebra
SpinSpace
Jminus
SpinOperator
Jz
Jplus
qnet.algebra.core.hilbert_space_algebra
LocalSpace
HilbertSpace
ProductSpace
qnet.algebra.library.fock_operators
Create
Destroy
Displace
Phase
Squeeze
qnet.algebra.core.operator_algebra
Commutator
OperatorSymbol
ScalarTimesOperator
OperatorPlus
OperatorTimes
OperatorTrace
PseudoInverse
NullSpaceProjector
OperatorPlusMinusCC
Operator
LocalOperator
LocalSigma
OperatorDerivative
OperatorIndexedSum
Adjoint
qnet.algebra.core.state_algebra
KetPlus
TensorKet
StateDerivative
KetIndexedSum
Bra
qnet.algebra.core.abstract_quantum_algebra
QuantumOperation
QuantumPlus
QuantumDerivative
QuantumSymbol
QuantumTimes
QuantumIndexedSum
QuantumExpression
ScalarTimesQuantumExpression
SingleQuantumOperation
QuantumAdjoint
KetBra
State
LocalKet
CoherentStateKet
KetSymbol
BasisKet
ScalarTimesKet
OperatorTimesKet
BraKet
qnet.algebra.core.super_operator_algebra
SuperOperatorDerivative
SuperAdjoint
SuperOperatorPlus
SuperOperatorTimes
SuperOperator
SuperOperatorSymbol
ScalarTimesSuperOperator
qnet.algebra.core.indexed_operations
IndexedSum
SuperOperatorTimesOperator
SPost
SPre
qnet.algebra.core.abstract_algebra
Expression
Operation

CHAPTER 6

Symbolic Algebra

6.1 Expressions and Operations

QNET includes a rich (and extensible) symbolic algebra system for quantum mechanics and circuit models. The foundation of the symbolic algebra are the `Expression` class and its subclass `Operation`.

A general algebraic expression has a tree structure. The branches of the tree are operations; their children are the operands. The leaves of the tree are scalars or "atomic" expressions, where "atomic" means *not* an object of type `Operation` (e.g., a symbol)

For example, the `KetPlus` operation defines the sum of Hilbert space vectors, represented as:

```
KetPlus(psi1, psi2, ..., psiN)
```

All operations follow this pattern:

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

where `Head` is a subclass of `Operation` and `op1 .. opN` are the operands, which may be other operations, scalars, or atomic `Expression` objects.

Note that all expressions (inluding operations) can have associated *arguments*. For example `KetSymbol` takes *label* as an argument, and the Hilbert space displacement operator `Displace` takes a displacement amplitude as an argument. To avoid confusion between operands and arguments, operations are required to take their operands as positional arguments, and possible additional arguments as keyword arguments.

Expressions should generally not be instantiated directly, but through their `create()` method allowing for simplifications. This is true both for operations and atomic expressions. For example, instantiating `Displace` with `alpha=0` results in an `IdentityOperator` (unlike direct instantiation, the create method of any class may or may not return an instance of the same class). For operations, the *create* method handles the application of algebraic rules such as associativity (translating e.g. `KetPlus(psi1, KetPlus(psi2, psi3))` into `KetPlus(psi1, psi2, psi3))`

Many operations are associated with infix operators, e.g. a `KetPlus` instance is automatically created if two instances of `KetSymbol` are added with `+`. In this case, the `create()` method is used automatically.

Expressions and Operations are considered immutable: any change to the expression tree (e.g. an algebraic simplification) generates a new expression.

6.1.1 Defining `Operation` subclasses

When extending an algebra with new operations, it is essential to define the expression rewriting ("simplification") rules that govern how new expressions are instantiated. To this end, the `_simplification` class attribute of an *`Expression`* subclass must be defined. This attribute contains a list of callables. Each of these callables takes three parameters (the class, the list `args` of positional arguments given to *`create()`* and a dictionary `kwargs` of keyword arguments given to *`create()`*) and return either a tuple of new `args` and `kwargs` (which are then handed to the next callable), or an *`Expression`* (which is directly returned as the result of the call to *`Expression.create()`*).

Callables such as as `assoc()`, `idem()`, `orderby()`, and `filter_neutral()` handle common algebraic properties such as associativity or commutativity. The `match_replace()` and `match_replace_binary()` callables are central to any more advanced simplification through pattern matching. They delegate to a list of *`Patterns`* and replacements that are defined in the `_rules`, respectively `_binary_rules` class attributes of the *`Expression`* subclass.

The pattern matching rules may temporarily extended or modified using the `qnet.algebra.toolbox.core.extra_rules()`, `qnet.algebra.toolbox.core.extra_binary_rules()`, and `qnet.algebra.toolbox.core.no_rules()` context managers.

6.1.2 Pattern matching

The application of patterns is central to symbolic algebra. Patterns are defined and applied using the classed and helper routines in the *`pattern_matching`* module.

There are two main places where pattern matching comes up:

- automatically, through `match_replace()` and `match_replace_binary()` simplifications applied inside of *`Expression.create()`*.
- manually, through the `simplify()` function (or the `Expression.simplify()` method)

Since inside `match_replace()` and `match_replace_binary()`, patterns are matched against expressions that are not yet instantiated (we call these *`ProtoExpressions`*), the patterns in the `_rules` and `_binary_rules` class attributes are always constructed using the *`pattern_head()`* helper function. In contrast, patterns for `simplify()` are usually created through the *`pattern()`* helper function. The *`wc()`* function is used to associate Expression arguments with wildcard names.

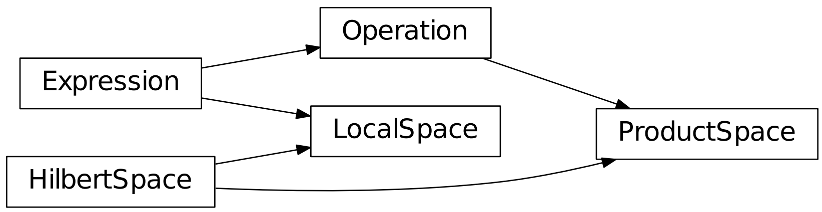
6.1.3 Algebraic Manipulations

While QNET automatically applies a large number of rules and simplifications if expressions are instantiated through the `create()` method, significant value is placed on manually manipulating algebraic expressions. In fact, this is one of the design considerations that separates it from the Sympy package: The rule-based transformations are both explicit and optional, allowing to instantiate expressions exactly in the desired form, and to apply specifc manipulations. Unlink in Sympy, the (tex) form of an expressions will directly reflect the structure of the expression, and the ordering of terms can be configured by the user. Thus, a Jupyter Notebook could document a symbolic derivation in the exact form one would normally write that derivation out by hand.

Common maniupulations and symbolic algorithms are collected in *`qnet.algebra.toolbox`*.

6.2 Hilbert Space Algebra

The hilbert_space_algebra module defines a simple algebra of finite dimensional or countably infinite dimensional Hilbert spaces.



Local/primitive degrees of freedom (e.g. a single multi-level atom or a cavity mode) are described by a LocalSpace; it requires a label, and may define a basis through the *basis* or *dimension* arguments. The LocalSpace may also define custom identifiers for operators acting on that space (subclasses of LocalOperator):

```
>>> a = Destroy(hs=1)
>>> ascii(a)
'a^(1)'
>>> hs1_custom = LocalSpace(1, local_identifiers={'Destroy': 'b'})
>>> b = Destroy(hs=hs1_custom)
>>> ascii(b)
'b^(1)'
```

Instances of LocalSpace combine via a product into composite tensor product spaces are given by instances of the ProductSpace

Furthermore,

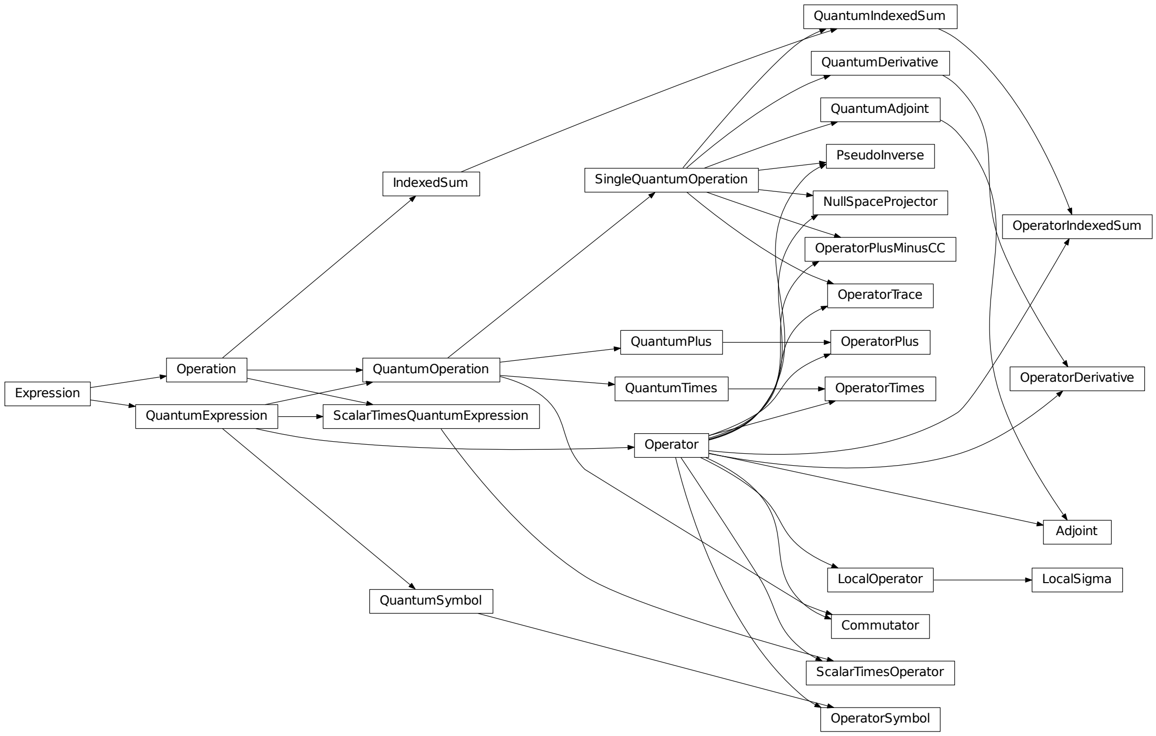
- the TrivialSpace represents a *trivial*[1] Hilbert space $\mathcal{H}_0 \simeq \mathbb{C}$
- the FullSpace represents a Hilbert space that includes all possible degrees of freedom.

Expressions in the operator, state, and superoperator algebra (discussed below) will all be associated with a Hilbert space. If any expressions are intended to be fed into a numerical simulation, all their associated Hilbert spaces must have a known dimension. Since all expressions are immutable, it is important to either define the all the LocalSpace instances they depend on with *basis* or *dimension* arguments first, or to later generate new expression with updated Hilbert spaces through the substitute() routine.

6.3 Operator Algebra

The operator_algebra module implements and algebra of Hilbert space operators

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



Operator expressions are constructed from sums (*OperatorPlus*) and products (*OperatorTimes*) of some basic elements, most importantly *local* operators (subclasses of *LocalOperator*). This include some very common symbolic operator such as

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

Furthermore, there exist symbolic representations for constants and symbols:

- the *IdentityOperator*
- the *ZeroOperator*
- an arbitrary *OperatorSymbol*

There are also a number of algebraic operations that act only on a single operator as their only operand. These include:

- the Hilbert space *Adjoint* operator $X^\dagger$
- *PseudoInverse* of operators X^+ satisfying $XX^+X = X$ and $X^+XX^+ = X^+$ as well as $(X^+X)^\dagger = X^+X$ and $(XX^+)^\dagger = XX^+$
- the kernel projection operator (*NullSpaceProjector*) $\mathcal{P}_{\mathrm{Ker}X}$ satisfying both $X\mathcal{P}_{\mathrm{Ker}X} = 0$ and $X^+X = 1 - \mathcal{P}_{\mathrm{Ker}X}$
- Partial traces over Operators $\mathrm{Tr}_s X$: *OperatorTrace*

6.3.1 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:

```
>>> from sympy import I
>>> def H_JC(Delta, Theta, epsilon, g):
...
...     # create Fock- and Atom local spaces
...     fock = LocalSpace('fock')
...     tls = LocalSpace('tls', basis=('e', 'g'))
...
...     # create representations of a and sigma
...     a = Destroy(hs=fock)
...     sigma = LocalSigma('g', 'e', hs=tls)
...
...     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_JC(Delta, Theta, epsilon, g)
>>> H
i ϵ (-a^(fock)† + a) + Θ a^(fock)† a + i g (a^(fock)† |ge| - a |eg|) + Δ |ee|

>>> H.space
_fock  _tls
```

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

```
>>> Create(hs=2) * Create(hs=1)
a^(1)† a^(2)†
```

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

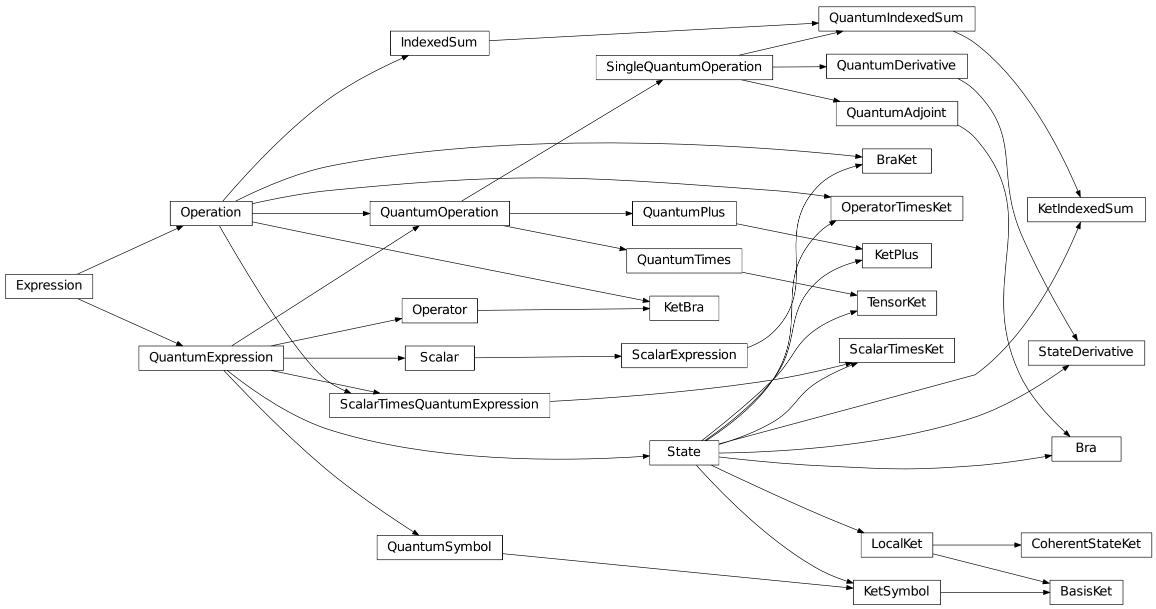
```
>>> Destroy(hs=1) * Create(hs=1)
 + a^(1)† a¹
```

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

```
>>>  (Destroy(hs=1) * Destroy(hs=1) * Destroy(hs=1) * Create(hs=1) * Create(hs=1) *␣
↪Create(hs=1)).expand()
6 + a^(1)† a^(1)† a^(1)† a¹ a¹ a¹ + 9 a^(1)† a^(1)† a¹ a¹ + 18 a^(1)† a¹
```

6.4 State (Ket-) Algebra

The `state_algebra` module implements an algebra of Hilbert space states.



By default we represent states ψ as `Ket` vectors $\psi \to |\psi\rangle$. However, any state can also be represented in its adjoint *Bra* form, since those representations are dual:

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

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

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

Furthermore,

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

And conversely,

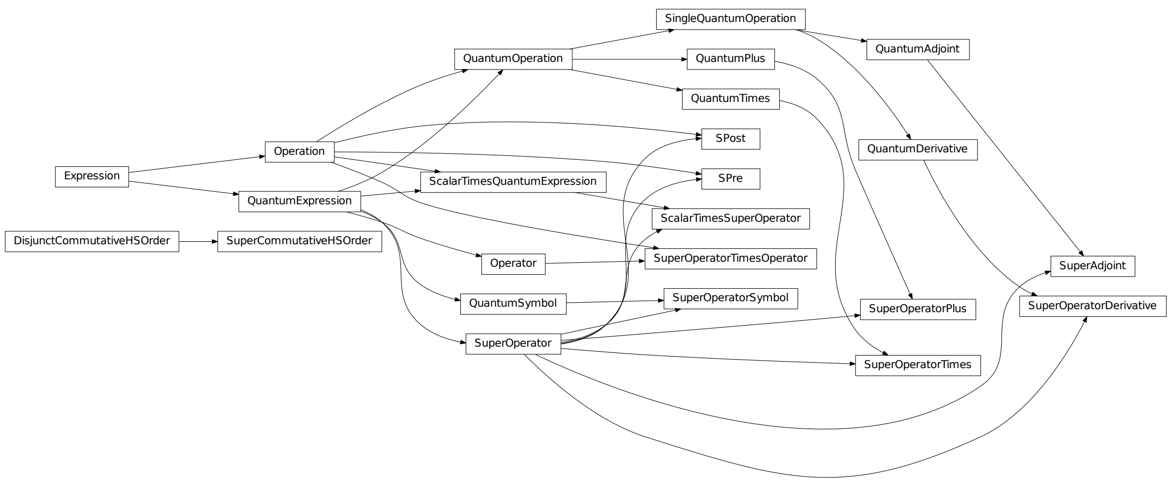
- a *Bra* can multiply a `Ket` from the left to create a (partial) inner product object *BraKet*. Currently, only full inner products are supported, i.e. the `Ket` and *Bra* operands need to have the same space.

There are also the following symbolic states:

- arbitrary *KetSymbols*
- the *TrivialKet* acting as the identity, and
- the *ZeroKet*.

6.5 Super-Operator Algebra

The `super_operator_algebra` contains an implementation of a superoperator algebra, i.e., operators acting on Hilbert space operator or elements of Liouville space (density matrices).



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(s, hs=FullSpace) for s in ("A", "B", "C"))
>>> 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
```

The neutral elements of super-operator addition and multiplication are `ZeroSuperOperator` and `IdentitySuperOperator`, respectively.

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.

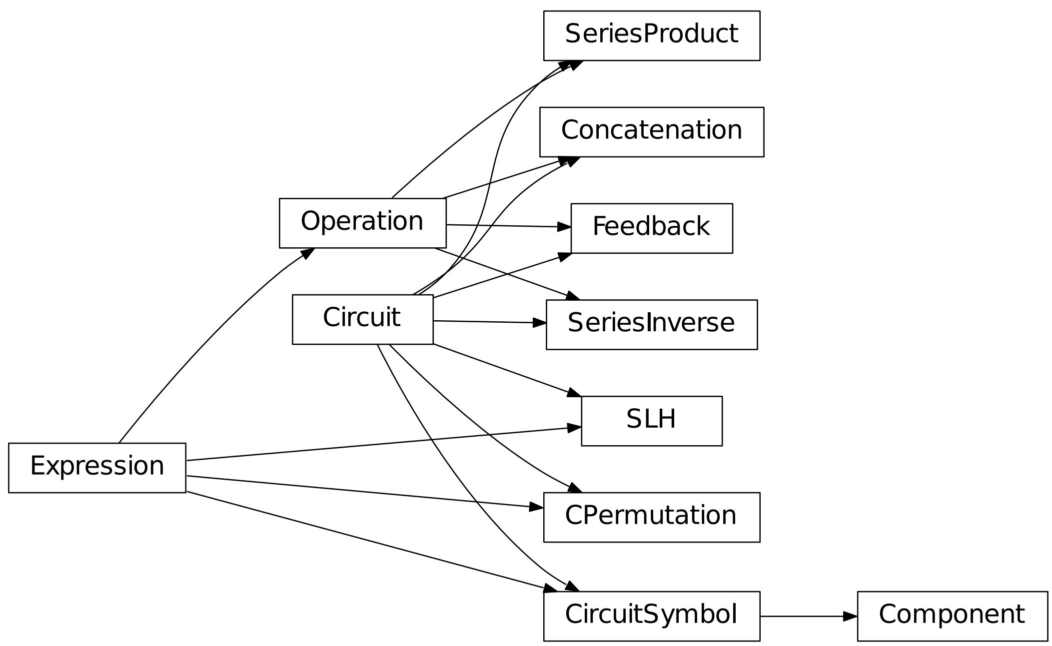
Super operators are applied to operators by multiplying an operator with superoperator from the left:

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

The result is an operator.

6.6 Circuit Algebra

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. This method is implemented in the `circuit_algebra` module.



A general system with an equal number n of input and output channels is described by the parameter triplet $(\mathbf{S}, \mathbf{L}, H)$, where H is the effective internal *Hamilton operator* for the system, $\mathbf{L} = (L_1, L_2, \dots, L_n)^T$ the *coupling vector* and $\mathbf{S} = (S_{jk})_{j,k=1}^n$ is the *scattering matrix* (whose elements are themselves operators). An element L_k of the coupling vector is given by a system operator that describes the system's coupling to the k-th input channel. Similarly, the elements S_{jk} of the scattering matrix are in general given by system operators describing the scattering between different field channels j and k.

The only conditions on the parameters are that the hamilton operator is self-adjoint and the scattering matrix is unitary:

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

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

The matrices of operators occuring in the SLH formalism are implemented in the `matrix_algebra` module.

6.6.1 Fundamental Circuit Operations

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

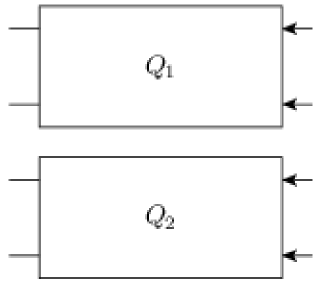


Fig. 1: $Q_1 \boxplus Q_2$

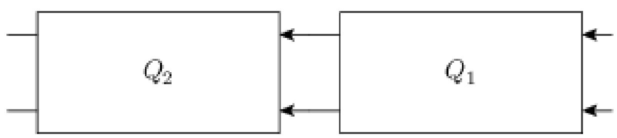


Fig. 2: $Q_2 \lhd Q_1$

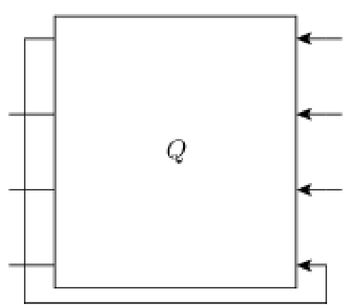


Fig. 3: $[Q]_{1\to 4}$

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.

6.6.2 Representation as Python objects

Python objects that are of the `Circuit` type have some of their operators overloaded to realize symbolic circuit algebra operations:

```
>>> A = CircuitSymbol('A', cdim=2)
>>> B = CircuitSymbol('B', cdim=2)
>>> print(srepr(A << B, cache={A: 'A', B: 'B'}))
SeriesProduct(A, B)
>>> print(srepr(A + B, cache={A: 'A', B: 'B'}))
Concatenation(A, B)
>>> print(srepr(FB(A, out_port=0, in_port=1), cache={A: 'A'}))
Feedback(A, out_port=0, in_port=1)
```

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

6.6.3 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, n=0):
...
...     # create Fock- and Atom local spaces
...     fock = LocalSpace('fock_%s' % n)
...     tls = LocalSpace('tls_%s' % n, basis=('e', 'g'))
...
...     # create representations of a and sigma
...     a = Destroy(hs=fock)
...     sigma = LocalSigma('g', 'e', hs=tls)
...
...     # 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
```

(continues on next page)

(continued from previous page)

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

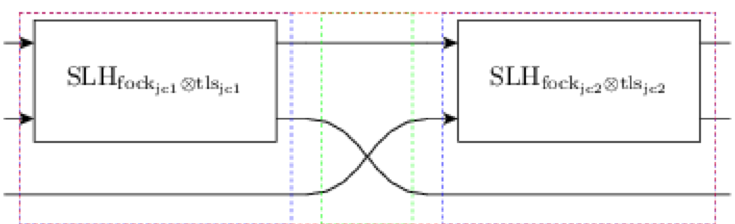
Consider now an example where we feed one Jaynes-Cummings system's output into a second one:

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

>>> JC1 = SLH_JaynesCummings(Delta, Theta, epsilon, g, kappa, gamma, n=1)
>>> JC2 = SLH_JaynesCummings(Delta, Theta, epsilon, g, kappa, gamma, n=2)

>>> from qnet import circuit_identity as cid
>>> SYS = (JC2 + cid(1)) << CPermutation((0, 2, 1)) << (JC1 + cid(1))
```

The resulting system's block diagram is:



and its overall SLH model is given by:

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

CHAPTER 7

Properties and Simplification of Circuit Algebraic Expressions

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

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

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

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

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

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

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

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

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

The following figures demonstrate the ambiguity of the circuit algebra:

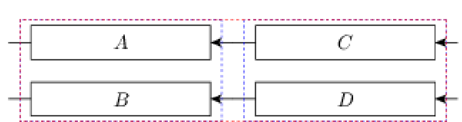


Fig. 1: $(A \boxplus B) \lhd (C \boxplus D)$

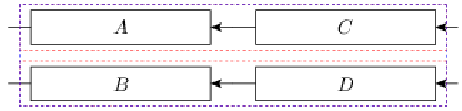


Fig. 2: $(A \lhd C) \boxplus (B \lhd D)$

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 $Q = (S, L, H)$ admits a decomposition into $1 \leq N \leq \operatorname{cdim} Q$ irreducible subsystems $Q = Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N$, where their channel dimensions satisfy $\operatorname{cdim} Q_j \geq 1,\ j = 1, 2, \dots N$ and $\sum_{j=1}^N \operatorname{cdim} Q_j = \operatorname{cdim} Q$. While their individual parameter triplets themselves are not uniquely determinedfootnote{Actually the scattering matrices $\{S_j\}$ and the coupling vectors $\{L_j\}$ *are* uniquely determined, but the Hamiltonian parameters $\{H_j\}$ must only obey the constraint $\sum_{j=1}^N H_j = H$.}, the sequence of their channel dimensions $(\operatorname{cdim} Q_1, \operatorname{cdim} Q_2, \dots \operatorname{cdim} Q_N) =: \operatorname{bls} Q$ 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 $\operatorname{bls} A = \operatorname{bls} B = (n_1, ...n_N)$, 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 $\operatorname{cdim} A_j = \operatorname{cdim} B_j = n_j,\ j = 1, \dots N$. 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).

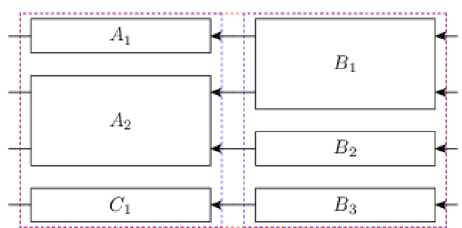


Fig. 3: Series "$(1, 2, 1) \lhd (2, 1, 1)$"

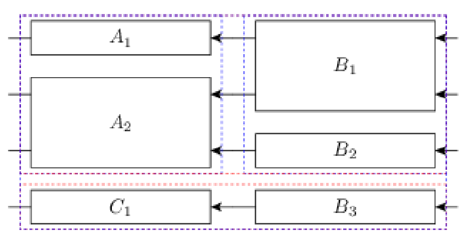


Fig. 4: Optimal decomposition into $(3, 1)$

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.

7.1 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 P, the system described by $(P, 0, 0)$ represents a pure permutation of the optical fields (ref fig permutation).

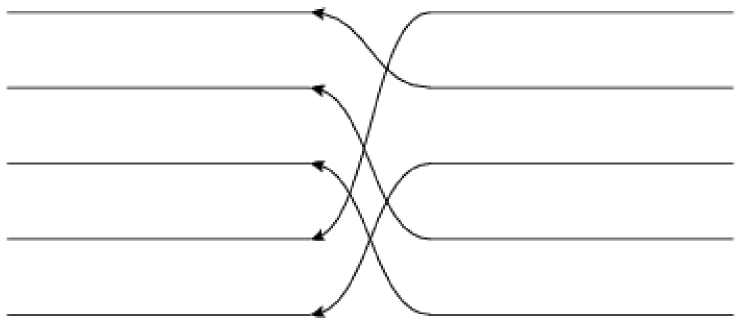


Fig. 5: A graphical representation of P_σ where $\sigma \equiv (4, 1, 5, 2, 3)$ in image tuple notation.

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

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

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

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

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

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

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

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

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

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

7.2 Permutations and Concatenations

Given a series $P_\sigma \lhd (Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N)$ 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 $(Q_1 \boxplus Q_2 \boxplus \cdots \boxplus Q_N) \lhd P_\sigma$, but this second scenario is closely relatedfootnote{Series-Inverting a series product expression also results in an inverted order of the operand inverses $(Q_1 \lhd Q_2)^{\lhd -1} = Q_2^{\lhd -1} \lhd Q_1^{\lhd -1}$. 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

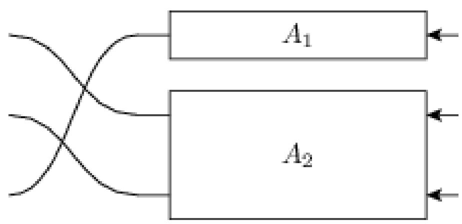


Fig. 6: $P_\sigma \lhd (A_1 \boxplus A_2)$

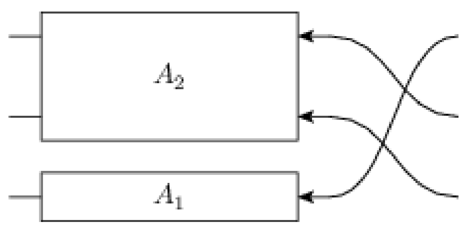


Fig. 7: $(A_2 \boxplus A_1) \lhd P_\sigma$

A block permuting series.

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

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

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

Block-factorizing permutations

The next-to-simplest case is realized when a permutation σ can be decomposed $\sigma = \sigma_{\rm b} \circ \sigma_{\rm i}$ into a permutation $\sigma_{\rm b}$ that block permutes the block structure n and an internal permutation $\sigma_{\rm i}$ that only permutes within each block, i.e.~:math:*sigma_{rm i} = sigma_1 boxplus sigma_2 boxplus dots boxplus sigma_N*. In this case we can perform the following simplifications

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

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

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

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

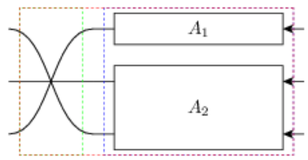


Fig. 8: $P_\sigma \lhd (A_1 \boxplus A_2)$

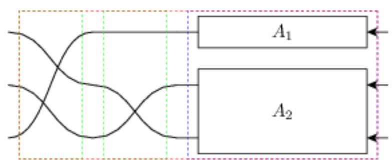


Fig. 9: $P_{\sigma_b} \lhd P_{\sigma_i} \lhd (A_1 \boxplus A_2)$

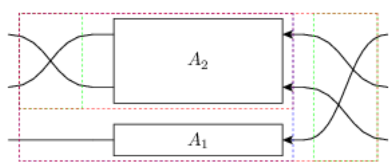


Fig. 10: $((P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_{\rm b}}$

A block factorizable series.

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

The general case

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

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

Hence, by construction, $\sigma_{\rm b} \circ \sigma_{\rm i}$ factorizes according to n so only $\sigma_{\rm x}$ remains on the exterior LHS of the expression.

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

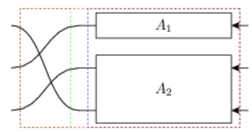


Fig. 11: $P_\sigma \lhd (A_1 \boxplus A_2)$

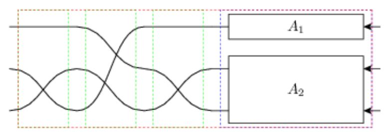


Fig. 12: $P_{\sigma_{\rm x}} \lhd P_{\sigma_{\rm b}} \lhd P_{\sigma_{\rm i}} \lhd (A_1 \boxplus A_2)$

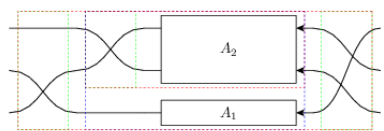


Fig. 13: $(P_{\sigma_{\rm x}} \lhd (P_{\sigma_2} \lhd A_2) \boxplus A_1) \lhd P_{\sigma_{\rm b}}$

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

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

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

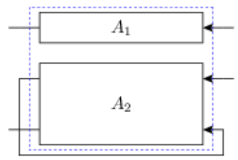


Fig. 14: $[A_1 \boxplus A_2]_{2\to 3}$

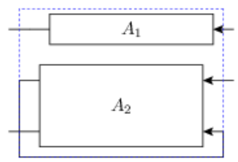


Fig. 15: $A_1 \boxplus [A_2]_{1\to 2}$

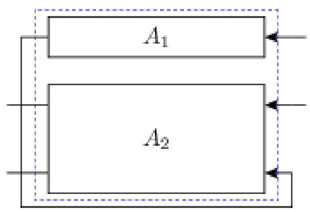


Fig. 16: $[A_1 \boxplus A_2]_{1\to 3}$

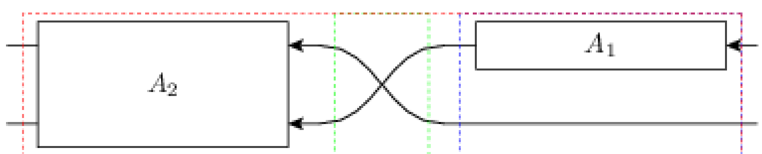


Fig. 17: $A_2 \lhd W_{2\leftarrow 1}^{(2)} \lhd (A_2 \boxplus \mathrm{id}_1)$

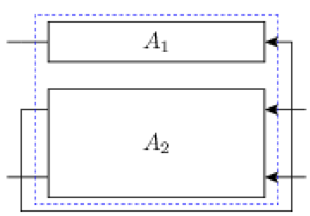


Fig. 18: $[A_1 \boxplus A_2]_{2\to 1}$

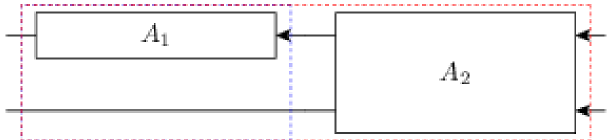


Fig. 19: $(A_1 \boxplus \mathrm{id}_1) \lhd A_2$

- $k, l \leq n_A$: 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).

7.4 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.

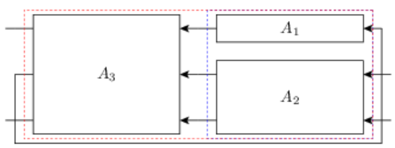


Fig. 20: $[A_3 \lhd (A_1 \boxplus A_2)]_{2\to 1}$

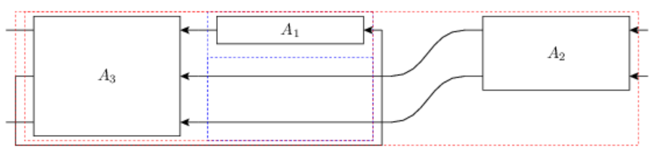


Fig. 21: $(A_3 \lhd (A_1 \boxplus \mathrm{id}_2)) \lhd A_2$

Reduction of series feedback with a concatenation at the RHS

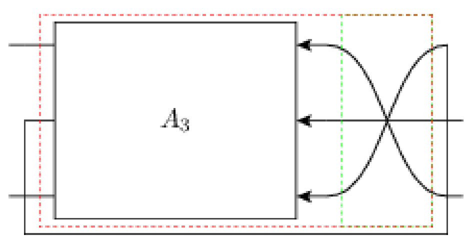


Fig. 22: $[A_3 \lhd P_\sigma]_{2\to 1}$

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 $(\mathrm{id}n_C - 1 \boxplus D) \lhd [(C \boxplus \mathrm{id}n_D) \lhd E]_{k\to l}$.

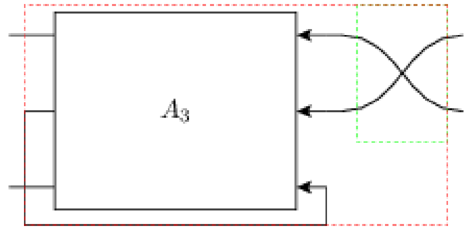


Fig. 23: $[A_3]_{2\to 3} \lhd P_{\tilde{\sigma}}$

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

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

Turning our attention to the bracketed expression within the feedback, we clearly see that it must be a permutation system $P_{\sigma'} = W_{n\leftarrow\sigma(l)}^{(n)} \lhd P_\sigma \lhd W_{l\leftarrow n}^{(n)}$ that maps $n \to l \to \sigma(l) \to n$. We can therefore write $\sigma' = \tilde{\sigma} \boxplus \sigma_{\mathrm{id}_1}$ 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:

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

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

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

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

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

CHAPTER 8

The Printing System

8.1 Overview

As a computer algebra framework, QNET puts great emphasis on the appropriate display of expressions, both in the context of a Jupyter notebook (QNETs main "graphical interface") and in the terminal. It also provides the possibility for you to completely customize the display.

The printing system is modeled closely after the printing system of SymPy (and directly builds on it). Unlike SymPy, however, the display of an expression will always directly reflect the algebraic structure (summands will not be re-ordered, for example).

In the context of a Jupyter notebook, expressions will be shown via LaTeX. In an interactive (I)Python terminal, a unicode rendering will be used if the terminal has unicode support, with a fallback to ascii. We can force this manually by:

```
>>> init_printing(repr_format='unicode')

>>> Create(hs='q_1') * CoherentStateKet(symbols('eta')**2/2, hs='q_1')
a^(q₁)† |α=η²/2^(q₁)
```

These textual renderings can be obtained manually through the `ascii()` and `unicode()` functions.

Unlike SymPy, the unicode rendering will not span multiple lines. Also, QNET will not rationalize the denominators of scalar fractions by default, to match the standard notation in quantum mechanics:

```
>>> (BasisKet(0, hs=1) + BasisKet(1, hs=1)) / sqrt(2)
1/2 (|0¹ + |1¹)
```

Compare this to the default in SymPy:

```
>>> (symbols('a') + symbols('b')) / sqrt(2)
2(a + b)
────────
    2
```

With the default settings, the LaTeX renderer that produces the output in the Jupyter notebook uses only tex macros that MathJax understands. You can obtain the LaTeX code through the `latex()` function. When generating code for a paper or report, it is better to customize the output for better readability with a more semantic use of macros, e.g. as:

```
>>> print(latex((BasisKet(0, hs=1) + BasisKet(1, hs=1)) / sqrt(2), tex_use_
↪braket=True))
\frac{1}{\sqrt{2}} \left(\Ket{0}^{(1)} + \Ket{1}^{(1)}\right)
```

In addition to the "mathematical" display of expressions, QNET also has functions to show the exact internal (tree) structure of an expression, either for debugging or for designing algebraic transformations.

The `srepr()` function returns the most direct representation of the expression: it is a string (possibly with indentation for the tree structure) that if evaluated results in the exact same expression.

An alternative, specifically for interactive use, is the `print_tree()` function. To generate a graphic representation of the tree structure, the `dotprint()` function produces a graph in the DOT language.

8.2 Basic Customization

At the beginning of an interactive session or notebook, the `init_printing()` routine should be called. This routine associates specific printing functions, e.g. `unicode()`, with the `__str__` and `__repr__` representation of an expression. This is what is returned by `str(expr)`, and by `repr(expr)` or as the output in an interactive (I)Python session. The initialization also specifies the default settings for each printing function. For example, you could suppress the display of Hilbert space labels:

```
>>> init_printing(show_hs_label=False, repr_format='unicode')
>>> (BasisKet(0, hs=1) + BasisKet(1, hs=1)) / sqrt(2)
1/2 (|0 + |1)
```

Or, in a debugging session, you could switch the default representation to use the indented `srepr()`:

```
>>> init_printing(repr_format='indsrepr')
>>> (BasisKet(0, hs=1) + BasisKet(1, hs=1)) / sqrt(2)
ScalarTimesKet(
    Mul(Rational(1, 2), Pow(Integer(2), Rational(1, 2))),
    KetPlus(
        BasisKet(
            0,
            hs=LocalSpace(
                '1')),
        BasisKet(
            1,
            hs=LocalSpace(
                '1'))))
```

The settings can also be changed *temporarily* via the `configure_printing()` context manager.

Note that `init_printing()` should only be called once; or else it should be given the `reset` parameter:

```
>>> init_printing(repr_format='unicode', reset=True)
```

8.3 Printer classes

The printing functions `ascii()`, `unicode()`, and `latex()` each delegate to an internal printer object that subclasses `qnet.printing.base.QnetBasePrinter`. After initialization, the printer class is referenced at e.g. `ascii.printer`.

For the ultimate control in customizing the printing system, you can implement your own subclasses of `QnetBasePrinter`, which is in turn a subclass of `sympy.printing.printer.Printer`. Thus, the overview of SymPy's printing system applies.

The QNET printers conceptually extend SymPy printers in the following ways:

- QNET printers have support for caching. One reason for this is efficiency. More importantly, it allows to pass a pre-initialized cache to force certain expressions to be represented by fixed strings, which can make expressions considerably more readable, and aids in generating code from expressions, see the example for `srepr()` .
- Every printer contains a sub-printer in the *_sympy_printer* attribute, instantiated from the *sympy_printer_cls* class attribute. Actual SymPy objects (e.g., scalar coefficients) are delegated to this sub-printer, while the main printer handles all `Expression` instances. Not that the default sub-printers use classes from `qnet.printing.sympy` that implement some custom printing more in line with the conventions of quantum physics.

When `init_printing()` is called with direct settings as in the previous section, these will be used as *global* settings, and will affect any printers (including SymPy sub-printers) that are instantiated afterwards.

The settings that are given to any printing function will be used for that specific call of the printing function only. If you define custom classes with different or additional settings and set them up for use with the printing function (see below), the accepted arguments to the printing functions change accordingly.

8.4 Customization through an INI file

While `init_printing()` can simply be called with explicit settings to configure the printing system globally (see above), for a more advanced set up an INI-file can be used. In this case, the path to the file must be the only argument:

```
init_printing(inifile=<path to file>)
```

This allows to associate custom printer classes with the printing functions, and also define the settings settings for those particular printers (as opposed to just global settings).

The INI file may have sections 'global', 'ascii', 'unicode', and 'latex'. Parameters in the 'global' section are equivalent to those could be passed to `init_printing()` as direct settings. That is, they set up the printing function to be used for `__str__` and `__repr__`, and set the global options for all printer classes.

The 'ascii', 'unicode', and 'latex' sections configure the respective printing functions. To link them to custom Printer classes, you may specify `printer` and `sympy_printer` as the full path to the Printer class that should be used for the main printer and the sub-printer for SymPy expressions. All other settings in the sections override the settings from 'global' for that particular printer.

Consider the following annotated example for an INI file:

```
[global]
# The settings in the 'global' section are for all Printer classes (both
# SymPy and QNET). They are equivalent to passing them to init_printing
# directly

# the printing function to use for str(expr)
```

(continues on next page)

(continued from previous page)

```
str_format = ascii
# the printing function to use for expr(expr)
repr_format = unicode
# direct global settings
show_hs_label = False
sig_as_ketbra = False
# note that boolean values must be specified as "True", or "False"

# The three sections below associate the printing functions with particular
# Printer classes, and override the global settings for those particular
# printers

[ascii]
printer = qnet.printing.asciiprinter.QnetAsciiPrinter
# we use the SymPy StrPrinter here, instead of the default
# qnet.printing.sympy.SympyStrPrinter that is customized to not
# rationalize denominators
sympy_printer = sympy.printing.str.StrPrinter
# we override the the settings from the 'global' section
show_hs_label = True
sig_as_ketbra = True

[unicode]
printer = qnet.printing.unicodeprinter.QnetUnicodePrinter
sympy_printer = qnet.printing.sympy.SympyUnicodePrinter
show_hs_label = subscript
unicode_op_hats = False

[latex]
printer = qnet.printing.latexprinter.QnetLatexPrinter
sympy_printer = qnet.printing.sympy.SympyLatexPrinter
# string values can be written un-escaped
tex_op_macro = \Op{{{name}}}
tex_use_braket = True
# You can also include options for the sympy_printer
inv_trig_style = full
```

CHAPTER 9

API

9.1 qnet package

Main QNET package

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

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

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

QNET provides a "flat" API. That is, after

```
>>> import qnet
```

all submodules are directly accessible, e.g.

```
>>> qnet.algebra.core.operator_algebra.OperatorSymbol
<class 'qnet.algebra.core.operator_algebra.OperatorSymbol'>
```

Furthermore, every package exports the "public" symbols of any of its submodules/subpackages (public symbols are those listed in __all__)

```
>>> (qnet.algebra.core.operator_algebra.OperatorSymbol is
...  qnet.algebra.core.OperatorSymbol is qnet.algebra.OperatorSymbol is
...  qnet.OperatorSymbol)
True
```

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

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

may be useful.

Subpackages:

9.1.1 qnet.algebra package

Symbolic quantum and photonic circuit (SLH) algebra

Subpackages:

qnet.algebra.core package

The fundamental object hiearchies that constitute QNET's various algebras

Submodules:

qnet.algebra.core.abstract_algebra module

Base classes for all Expressions and Operations.

The abstract algebra package provides the foundation for symbolic algebra of quantum objects or circuits. All symbolic objects are an instance of `Expression`. Algebraic combinations of atomic expressions are instances of `Operation`. In this way, any symbolic expression is a tree of operations, with children of each node defined through the `Operation.operands` attribute, and the leaves being atomic expressions.

See *Expressions and Operations* for design details and usage.

Summary

Classes:

`Expression`	Base class for all QNET Expressions
`Operation`	Base class for "operations"

Functions:

`substitute`	Substitute symbols or (sub-)expressions with the given replacements and re-evalute the result

`__all__`: `Expression`, `Operation`, `substitute`

Reference

class qnet.algebra.core.abstract_algebra.**Expression**(**args*, ***kwargs*)

Bases: `object`

Base class for all QNET Expressions

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

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

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

will return and object identical to *expr*.

Class Attributes

- **instance_caching** (*bool*) – Flag to indicate whether the `create()` class method should cache the instantiation of instances. If True, repeated calls to `create()` with the same arguments return instantly, instead of re-evaluating all simplifications and rules.
- **simplifications** (*list*) – List of callable simplifications that `create()` will use to process its positional and keyword arguments. Each callable must take three parameters (the class, the list *args* of positional arguments given to `create()` and a dictionary *kwargs* of keyword arguments given to `create()`) and return either a tuple of new *args* and *kwargs* (which are then handed to the next callable), or an `Expression` (which is directly returned as the result of the call to `create()`). The built-in available simplification callables are in `algebraic_properties`

simplifications = []

instance_caching = True

classmethod create(**args*, ***kwargs*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

classmethod add_rule(*name*, *pattern*, *replacement*, *attr=None*)

Add an algebraic rule for `create()` to the class

Parameters

- **name** (`str`) – Name of the rule. This is used for debug logging to allow an analysis of which rules where applied when creating an expression. The *name* can be arbitrary, but it must be unique. Built-in rules have names `'Rxxx'` where `x` is a digit
- **pattern** (`Pattern`) – A pattern constructed by `pattern_head()` to match a `ProtoExpr`
- **replacement** (`callable`) – callable that takes the wildcard names defined in *pattern* as keyword arguments and returns an evaluated expression.
- **attr** (`None or str`) – Name of the class attribute to which to add the rule. If None, one of `'_rules'`, `'_binary_rules'` is automatically chosen

Raises

- `TypeError` – if *name* is not a `str` or *pattern* is not a `Pattern` instance
- `ValueError` – if *pattern* is not set up to match a `ProtoExpr`; if there there is already a rule with the same *name*; if *replacement* is not a callable or does not take all the wildcard names in *pattern* as arguments

- `AttributeError` – If invalid *attr*

Note: The "automatic" rules added by this method are applied *before* expressions are instantiated (against a corresponding `ProtoExpr`). In contrast, `apply_rules()`/`apply_rule()` are applied to fully instantiated objects.

The `temporary_rules()` context manager may be used to create a context in which rules may be defined locally.

---

classmethod show_rules(**names*, *attr=None*)

Print algebraic rules used by `create`

Print a summary of the algebraic rules with the given names, or all rules if not names a given.

Parameters

- **names** (`str`) – Names of rules to show
- **attr** (`None or str`) – Name of the class attribute from which to get the rules. Cf. `add_rule()`.

Raises `AttributeError` – If invalid *attr*

classmethod del_rules(**names*, *attr=None*)

Delete algebraic rules used by `create()`

Remove the rules with the given *names*, or all rules if no names are given

Parameters

- **names** (`str`) – Names of rules to delete
- **attr** (`None or str`) – Name of the class attribute from which to delete the rules. Cf. `add_rule()`.

Raises

- `KeyError` – If any rules in *names* does not exist
- `AttributeError` – If invalid *attr*

classmethod rules(*attr=None*)

Iterable of rule names used by `create()`

Parameters **attr** (`None or str`) – Name of the class attribute to which to get the names. If None, one of `'_rules'`, `'_binary_rules'` is automatically chosen

args

The tuple of positional arguments for the instantiation of the Expression

kwargs

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

minimal_kwargs

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

substitute(*var_map*)

Substitute sub-expressions

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

doit (*classes=None*, *recursive=True*, ***kwargs*)
Rewrite (sub-)expressions in a more explicit form

Return a modified expression that is more explicit than the original expression. The definition of "more explicit" is decided by the relevant subclass, e.g. a `Commutator` is written out according to its definition.

Parameters

- **classes** (*None or list*) – an optional list of classes. If given, only (sub-)expressions that an instance of one of the classes in the list will be rewritten.
- **recursive** (*bool*) – If True, also rewrite any sub-expressions of any rewritten expression. Note that `doit()` always recurses into sub-expressions of expressions not affected by it.
- **kwargs** – Any remaining keyword arguments may be used by the `doit()` method of a particular expression.

Example

Consider the following expression:

```
>>> from sympy import IndexedBase
>>> i = IdxSym('i'); N = symbols('N')
>>> Asym, Csym = symbols('A, C', cls=IndexedBase)
>>> A = lambda i: OperatorSymbol(StrLabel(Asym[i]), hs=0)
>>> B = OperatorSymbol('B', hs=0)
>>> C = lambda i: OperatorSymbol(StrLabel(Csym[i]), hs=0)
>>> def show(expr):
...     print(unicode(expr, show_hs_label=False))
>>> expr = Sum(i, 1, 3)(Commutator(A(i), B) + C(i)) / N
>>> show(expr)
1/N (_{i=1}^{3} (C_i + [A_i, B]))
```

Calling `doit()` without parameters rewrites both the indexed sum and the commutator:

```
>>> show(expr.doit())
1/N (C₁ + C₂ + C₃ + A₁ B + A₂ B + A₃ B - B A₁ - B A₂ - B A₃)
```

A non-recursive call only expands the sum, as it does not recurse into the expanded summands:

```
>>> show(expr.doit(recursive=False))
1/N (C₁ + C₂ + C₃ + [A₁, B] + [A₂, B] + [A₃, B])
```

We can selectively expand only the sum or only the commutator:

```
>>> show(expr.doit(classes=[IndexedSum]))
1/N (C₁ + C₂ + C₃ + [A₁, B] + [A₂, B] + [A₃, B])

>>> show(expr.doit(classes=[Commutator]))
1/N (_{i=1}^{3} (C_i - B A_i + A_i B))
```

Also we can pass a keyword argument that expands the sum only to the 2nd term, as documented in `Commutator.doit()`

```
>>> show(expr.doit(classes=[IndexedSum], max_terms=2))
1/N (C₁ + C₂ + [A₁, B] + [A₂, B])
```

apply (*func*, **args*, ***kwargs*)

Apply *func* to expression.

Equivalent to `func(self, *args, **kwargs)`. This method exists for easy chaining:

```
>>> A, B, C, D = (
...     OperatorSymbol(s, hs=1) for s in ('A', 'B', 'C', 'D'))
>>> expr = (
...     Commutator(A * B, C * D)
...     .apply(lambda expr: expr**2)
...     .apply(expand_commutators_leibniz, expand_expr=False)
...     .substitute({A: IdentityOperator}))
```

apply_rules (*rules*, *recursive=True*)

Rebuild the expression while applying a list of rules

The rules are applied against the instantiated expression, and any sub-expressions if *recursive* is True. Rule application is best though of as a pattern-based substitution. This is different from the *automatic* rules that `create()` uses (see `add_rule()`), which are applied *before* expressions are instantiated.

Parameters

- **rules** (*list or OrderedDict*) – List of rules or dictionary mapping names to rules, where each rule is a tuple (`Pattern`, replacement callable), cf. `apply_rule()`
- **recursive** (*bool*) – If true (default), apply rules to all arguments and keyword arguments of the expression. Otherwise, only the expression itself will be re-instantiated.

If *rules* is a dictionary, the keys (rules names) are used only for debug logging, to allow an analysis of which rules lead to the final form of an expression.

apply_rule (*pattern*, *replacement*, *recursive=True*)

Apply a single rules to the expression

This is equivalent to `apply_rules()` with `rules=[(pattern, replacement)]`

Parameters

- **pattern** (`Pattern`) – A pattern containing one or more wildcards
- **replacement** (*callable*) – A callable that takes the wildcard names in *pattern* as keyword arguments, and returns a replacement for any expression that *pattern* matches.

Example

Consider the following Heisenberg Hamiltonian:

```
>>> tls = SpinSpace(label='s', spin='1/2')
>>> i, j, n = symbols('i, j, n', cls=IdxSym)
>>> J = symbols('J', cls=sympy.IndexedBase)
>>> def Sig(i):
...     return OperatorSymbol(
...         StrLabel(sympy.Indexed('sigma', i)), hs=tls)
>>> H = - Sum(i, tls)(Sum(j, tls)(
...     J[i, j] * Sig(i) * Sig(j)))
>>> unicode(H)
'- (_{i,j  } J_ij σ_i^(s) σ_j^(s))'
```

We can transform this into a classical Hamiltonian by replacing the operators with scalars:

```
>>> H_classical = H.apply_rule(
...     pattern(OperatorSymbol, wc('label', head=StrLabel)),
...     lambda label: label.expr * IdentityOperator)
>>> unicode(H_classical)
'- (_{i,j  } J_ij σ_i σ_j)'
```

rebuild()
Recursively re-instantiate the expression

This is generally used within a managed context such as `extra_rules()`, `extra_binary_rules()`, or `no_rules()`.

free_symbols
Set of free SymPy symbols contained within the expression.

bound_symbols
Set of bound SymPy symbols in the expression

all_symbols
Combination of *free_symbols* and *bound_symbols*

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

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

Parameters

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

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

Base class for "operations"

Operations are Expressions that act algebraically on other expressions (their "operands").

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

operands
Tuple of operands of the operation

args
Alias for operands

qnet.algebra.core.abstract_quantum_algebra module

Common algebra of "quantum" objects

Quantum objects have an associated Hilbert space, and they (at least partially) summation, products, multiplication with a scalar, and adjoints.

The algebra defined in this module is the superset of the Hilbert space algebra of states (augmented by the tensor product), and the C* algebras of operators and superoperators.

Summary

Classes:

QuantumAdjoint	Base class for adjoints of quantum expressions
QuantumDerivative	Symbolic partial derivative
QuantumExpression	Base class for expressions associated with a Hilbert space
QuantumIndexedSum	Base class for indexed sums
QuantumOperation	Base class for operations on quantum expression
QuantumPlus	General implementation of addition of quantum expressions
QuantumSymbol	Symbolic element of an algebra
QuantumTimes	General implementation of product of quantum expressions
ScalarTimesQuantumExpression	Product of a *Scalar* and a *QuantumExpression*
SingleQuantumOperation	Base class for operations on a single quantum expression

Functions:

Sum	Instantiator for an arbitrary indexed sum.
ensure_local_space	Ensure that the given *hs* is an instance of `LocalSpace`.

`__all__`: *QuantumAdjoint*, *QuantumDerivative*, *QuantumExpression*, *QuantumIndexedSum*, *QuantumOperation*, *QuantumPlus*, *QuantumSymbol*, *QuantumTimes*, *ScalarTimesQuantumExpression*, *SingleQuantumOperation*, *Sum*

Reference

class qnet.algebra.core.abstract_quantum_algebra.**QuantumExpression**(**args*, ***kwargs*)

Bases: *qnet.algebra.core.abstract_algebra.Expression*

Base class for expressions associated with a Hilbert space

is_zero

Check whether the expression is equal to zero.

Specifically, this checks whether the expression is equal to the neutral element for the addition within the algebra. This does not generally imply equality with a scalar zero:

```
>>> ZeroOperator.is_zero
True
>>> ZeroOperator == 0
False
```

space

The *HilbertSpace* on which the operator acts non-trivially

adjoint()

The Hermitian adjoint of the Expression

dag()
Alias for *adjoint()*

expand()
Expand out distributively all products of sums.

Note: This does not expand out sums of scalar coefficients. You may use *simplify_scalar()* for this purpose.

simplify_scalar(*func=<function simplify>*)
Simplify all scalar symbolic (SymPy) coefficients by appyling *func* to them

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

Parameters

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

Returns The n-th derivative.

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

When expanding an expr for a parameter x about the point x_0 up to order N, the resulting coefficients $(c_1, \dots, c_N)$ fulfill

$$\text{expr} = \sum_{n=0}^{N} c_n (x - x_0)^n + O(N+1)$$

Parameters

- **param** (Symbol) – Expansion parameter x
- **about** (Scalar) – Point x_0 about which to expand
- **order** (int) – Maximum order N of expansion (>= 0)

Return type tuple

Returns tuple of length `order + 1`, where the entries are the expansion coefficients, $(c_0, \dots, c_N)$.

Note: The expansion coefficients are "type-stable", in that they share a common base class with the original expression. In particular, this applies to "zero" coefficients:

```
>>> expr = KetSymbol("Psi", hs=0)
>>> t = sympy.symbols("t")
>>> assert expr.series_expand(t, 0, 1) == (expr, ZeroKet)
```

class qnet.algebra.core.abstract_quantum_algebra.**QuantumSymbol**(*label, *sym_args, hs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumExpression*

Symbolic element of an algebra

Parameters

- **label** (*str or SymbolicLabelBase*) – Label for the symbol
- **sym_args** (Scalar) – optional scalar arguments. With zero *sym_args*, the resulting symbol is a constant. With one or more *sym_args*, it becomes a function.
- **hs** (HilbertSpace, *str*, *int*, *tuple*) – the Hilbert space associated with the symbol. If a *str* or an *int*, an implicit (sub-)instance of `LocalSpace` with a corresponding label will be created, or, for a tuple of *str* or *int*, a `ProducSpace. The type of the implicit Hilbert space is set by :func:`.init_algebra`.

label
Label of the symbol

args
Tuple of positional arguments, consisting of the label and possible *sym_args*

kwargs
Dict of keyword arguments, containing only *hs*

sym_args
Tuple of scalar arguments of the symbol

space
The *HilbertSpace* on which the operator acts non-trivially

free_symbols
Set of free SymPy symbols contained within the expression.

class qnet.algebra.core.abstract_quantum_algebra.**QuantumOperation**(**operands*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumExpression*, *qnet.algebra.core.abstract_algebra.Operation*

Base class for operations on quantum expression

These are operations on quantum expressions within the same fundamental set.

space
Hilbert space of the operation result

class qnet.algebra.core.abstract_quantum_algebra.**SingleQuantumOperation**(*op*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumOperation*

Base class for operations on a single quantum expression

operand
The operator that the operation acts on

class qnet.algebra.core.abstract_quantum_algebra.**QuantumAdjoint**(*op*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*

Base class for adjoints of quantum expressions

class qnet.algebra.core.abstract_quantum_algebra.**QuantumPlus**(**operands*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumOperation*

General implementation of addition of quantum expressions

order_key
alias of *qnet.utils.ordering.FullCommutativeHSOrder*

class qnet.algebra.core.abstract_quantum_algebra.**QuantumTimes**(**operands*, ***kwargs*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumOperation*

General implementation of product of quantum expressions

order_key

alias of *qnet.utils.ordering.DisjunctCommutativeHSOrder*

factor_for_space(*spc*)

Return a tuple of two products, where the first product contains the given Hilbert space, and the second product is disjunct from it.

class qnet.algebra.core.abstract_quantum_algebra.**ScalarTimesQuantumExpression**(*coeff*, *term*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumExpression*, *qnet.algebra.core.abstract_algebra.Operation*

Product of a *Scalar* and a *QuantumExpression*

classmethod create(*coeff*, *term*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are *match_replace()* and *match_replace_binary()* which apply rule-based simplifications.

The *temporary_rules()* context manager may be used to allow temporary modification of the automatic simplifications that *create()* uses, in particular the rules for *match_replace()* and *match_replace_binary()*. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

coeff

term

free_symbols

Set of free SymPy symbols contained within the expression.

space

The *HilbertSpace* on which the operator acts non-trivially

class qnet.algebra.core.abstract_quantum_algebra.**QuantumDerivative**(*op*, *, *derivs*, *vals=None*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*

Symbolic partial derivative

$$\frac{\partial^n}{\partial x_1^{n_1} \dots \partial x_N^{n_N}} A(x_1, \dots, x_N); \qquad \text{with} \quad n = \sum_i n_i$$

Alternatively, if *vals* is given, a symbolic representation of the derivative (partially) evaluated at a specific point.

$$\left. \frac{\partial^n}{\partial x_1^{n_1} \dots \partial x_N^{n_N}} A(x_1, \dots, x_N) \right|_{x_1 = v_1, \dots}$$

Parameters

- **op** (`QuantumExpression`) – the expression $A(x_1, \dots, x_N)$ that is being derived

- **derivs** (*dict*) – a map of symbols x_i to the order n_i of the derivate with respect to that symbol
- **vals** (*dict or None*) – If not None, a map of symbols x_i to values v_i for the point at which the derivative should be evaluated.

Note: *QuantumDerivative* is intended to be instantiated only inside the `_diff()` method of a *QuantumExpression*, for expressions that depend on scalar arguments in an unspecified way. Generally, if a derivative can be calculated explicitly, the explicit form is preferred over the abstract *QuantumDerivative*.

simplifications = [<function derivative_via_diff>]

classmethod create (*op*, *, *derivs*, *vals=None*)
Instantiate the derivative by repeatedly calling the `_diff()` method of *op* and evaluating the result at the given *vals*.

kwargs
Keyword arguments for the instantiation of the derivative

minimal_kwargs
Minimal keyword arguments for the instantiation of the derivative (excluding defaults)

evaluate_at (*vals*)
Evaluate the derivative at a specific point

derivs
Mapping of symbols to the order of the derivative with respect to that symbol. Keys are ordered alphanumerically.

syms
Set of symbols with respect to which the derivative is taken

vals
Mapping of symbols to values for which the derivative is to be evaluated. Keys are ordered alphanumerically.

free_symbols
Set of free SymPy symbols contained within the expression.

bound_symbols
Set of Sympy symbols that are eliminated by evaluation.

n
The total order of the derivative.

This is the sum of the order values in *derivs*

class qnet.algebra.core.abstract_quantum_algebra.**QuantumIndexedSum** (*term*, **ranges*)
Bases: *qnet.algebra.core.indexed_operations.IndexedSum*, *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*

Base class for indexed sums

space
The Hilbert space of the sum's term

qnet.algebra.core.abstract_quantum_algebra.**Sum** (*idx*, **args*, ***kwargs*)
Instantiator for an arbitrary indexed sum.

This returns a function that instantiates the appropriate `QuantumIndexedSum` subclass for a given term expression. It is the preferred way to "manually" create indexed sum expressions, closely resembling the normal mathematical notation for sums.

Parameters

- **idx** (`IdxSym`) – The index symbol over which the sum runs
- **args** – arguments that describe the values over which *idx* runs,
- **kwargs** – keyword-arguments, used in addition to *args*

Returns an instantiator function that takes a arbitrary *term* that should generally contain the *idx* symbol, and returns an indexed sum over that *term* with the index range specified by the original *args* and *kwargs*.

Return type callable

There is considerable flexibility to specify concise *args* for a variety of index ranges.

Assume the following setup:

```
>>> i = IdxSym('i'); j = IdxSym('j')
>>> ket_i = BasisKet(FockIndex(i), hs=0)
>>> ket_j = BasisKet(FockIndex(j), hs=0)
>>> hs0 = LocalSpace('0')
```

Giving *i* as the only argument will sum over the indices of the basis states of the Hilbert space of *term*:

```
>>> s = Sum(i)(ket_i)
>>> unicode(s)
'_{i  ₀} |i⁰'
```

You may also specify a Hilbert space manually:

```
>>> Sum(i, hs0)(ket_i) == Sum(i, hs=hs0)(ket_i) == s
True
```

Note that using `Sum()` is vastly more readable than the equivalent "manual" instantiation:

```
>>> s == KetIndexedSum.create(ket_i, IndexOverFockSpace(i, hs=hs0))
True
```

By nesting calls to *Sum*, you can instantiate sums running over multiple indices:

```
>>> unicode( Sum(i)(Sum(j)(ket_i * ket_j.dag())) )
'_{i,j  ₀} |ij|⁰'
```

Giving two integers in addition to the index *i* in *args*, the index will run between the two values:

```
>>> unicode( Sum(i, 1, 10)(ket_i) )
'_{i=1}^{10} |i⁰'
>>> Sum(i, 1, 10)(ket_i) == Sum(i, 1, to=10)(ket_i)
True
```

You may also include an optional step width, either as a third integer or using the *step* keyword argument.

```
>>> #unicode( Sum(i, 1, 10, step=2)(ket_i) ) # TODO
```

Lastly, by passing a tuple or list of values, the index will run over all the elements in that tuple or list:

```
>>> unicode( Sum(i,  (1, 2, 3))(ket_i))
'_{i  {1,2,3}} |i^0'
```

qnet.algebra.core.abstract_quantum_algebra.**ensure_local_space**(*hs*, *cls=<class 'qnet.algebra.core.hilbert_space_algebra..*

Ensure that the given *hs* is an instance of `LocalSpace`.

If *hs* an instance of `str` or `int`, it will be converted to a *cls* (if possible). If it already is an instace of *cls*, *hs* will be returned unchanged.

Parameters

- **hs** (`HilbertSpace` *or* `str` *or* `int`) – The Hilbert space (or label) to convert/check
- **cls** (*type*) – The class to which an int/str label for a Hilbert space should be converted. Must be a subclass of `LocalSpace`.

Raises `TypeError` – If *hs* is not a *LocalSpace*, `str`, or `int`.

Returns original or converted *hs*

Return type *LocalSpace*

Examples

```
>>> srepr(ensure_local_space(0))
"LocalSpace('0')"
>>> srepr(ensure_local_space('tls'))
"LocalSpace('tls')"
>>> srepr(ensure_local_space(0, cls=LocalSpace))
"LocalSpace('0')"
>>> srepr(ensure_local_space(LocalSpace(0)))
"LocalSpace('0')"
>>> srepr(ensure_local_space(LocalSpace(0)))
"LocalSpace('0')"
>>> srepr(ensure_local_space(LocalSpace(0) * LocalSpace(1)))
Traceback (most recent call last):
   ...
TypeError: hs must be an instance of LocalSpace
```

qnet.algebra.core.algebraic_properties module

Summary

Functions:

accept_bras	Accept operands that are all bras, and turn that into to bra of the operation applied to all corresponding kets
assoc	Associatively expand out nested arguments of the flat class.
assoc_indexed	Flatten nested indexed structures while pulling out possible prefactors

Continued on next page

Table 5 – continued from previous page

basis_ket_zero_outside_hs	For `BasisKet.create(ind, hs)` with an integer label *ind*, return a `ZeroKet` if *ind* is outside of the range of the underlying Hilbert space
check_cdims	Check that all operands (*ops*) have equal channel dimension.
collect_scalar_summands	Collect `ValueScalar` and `ScalarExpression` summands
collect_summands	Collect summands that occur multiple times into a single summand
commutator_order	Apply anti-commutative property of the commutator to apply a standard ordering of the commutator arguments
convert_to_scalars	Convert any entry in *ops* that is not a *Scalar* instance into a *ScalarValue* instance
convert_to_spaces	For all operands that are merely of type str or int, substitute LocalSpace objects with corresponding labels: For a string, just itself, for an int, a string version of that int.
delegate_to_method	Create a simplification rule that delegates the instantiation to the method *mtd* of the operand (if defined)
derivative_via_diff	Implementation of the `QuantumDerivative.create()` interface via the use of `QuantumExpression._diff()`.
disjunct_hs_zero	Return ZeroOperator if all the operators in *ops* have a disjunct Hilbert space, or an unchanged *ops*, *kwargs* otherwise
empty_trivial	A ProductSpace of zero Hilbert spaces should yield the TrivialSpace
filter_cid	Remove occurrences of the *circuit_identity()* `cid(n)` for any `n`.
filter_neutral	Remove occurrences of a neutral element from the argument/operand list, if that list has at least two elements.
idem	Remove duplicate arguments and order them via the cls's order_key key object/function.
implied_local_space	Return a simplification that converts the positional argument *arg_index* from (str, int) to a subclass of *LocalSpace*, as well as any keyword argument with one of the given keys.
indexed_sum_over_const	Execute an indexed sum over a term that does not depend on the summation indices
indexed_sum_over_kronecker	Execute sums over KroneckerDelta prefactors
match_replace	Match and replace a full operand specification to a function that provides a replacement for the whole expression or raises a *CannotSimplify* exception.
match_replace_binary	Similar to func:*match_replace*, but for arbitrary length operations, such that each two pairs of subsequent operands are matched pairwise.
orderby	Re-order arguments via the class's `order_key` key object/function.
scalars_to_op	Convert any scalar α in *ops* into an operator $alpha identity$

Reference

qnet.algebra.core.algebraic_properties.**assoc**(*cls*, *ops*, *kwargs*)

Associatively expand out nested arguments of the flat class. E.g.:

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

qnet.algebra.core.algebraic_properties.**assoc_indexed**(*cls*, *ops*, *kwargs*)

Flatten nested indexed structures while pulling out possible prefactors

For example, for an *IndexedSum*:

$$\sum_j \left(a \sum_i \dots \right) = a \sum_{j,i} \dots$$

qnet.algebra.core.algebraic_properties.**idem**(*cls*, *ops*, *kwargs*)

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

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

qnet.algebra.core.algebraic_properties.**orderby**(*cls*, *ops*, *kwargs*)

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

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

qnet.algebra.core.algebraic_properties.**filter_neutral**(*cls*, *ops*, *kwargs*)

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

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

qnet.algebra.core.algebraic_properties.**collect_summands**(*cls*, *ops*, *kwargs*)

Collect summands that occur multiple times into a single summand

Also filters out zero-summands.

Example

```
>>> A, B, C = (OperatorSymbol(s, hs=0) for s in ('A', 'B', 'C'))
>>> collect_summands(
...     OperatorPlus, (A, B, C, ZeroOperator, 2 * A, B, -C) , {})
((3 * A^(0), 2 * B^(0)), {})
>>> collect_summands(OperatorPlus, (A, -A), {})
ZeroOperator
>>> collect_summands(OperatorPlus, (B, A, -B), {})
A^(0)
```

qnet.algebra.core.algebraic_properties.**collect_scalar_summands**(*cls*, *ops*, *kwargs*)

Collect `ValueScalar` and `ScalarExpression` summands

Example

```
>>> srepr(collect_scalar_summands(Scalar, (1, 2, 3), {}))
'ScalarValue(6)'
>>> collect_scalar_summands(Scalar, (1, 1, -1), {})
One
>>> collect_scalar_summands(Scalar, (1, -1), {})
Zero
```

```
>>> Psi = KetSymbol("Psi", hs=0)
>>> Phi = KetSymbol("Phi", hs=0)
>>> braket = BraKet.create(Psi, Phi)
```

```
>>> collect_scalar_summands(Scalar, (1, braket, -1), {})
<Psi|Phi>^(0)
>>> collect_scalar_summands(Scalar, (1, 2 * braket, 2, 2 * braket), {})
((3, 4 * <Psi|Phi>^(0)), {})
>>> collect_scalar_summands(Scalar, (2 * braket, -braket, -braket), {})
Zero
```

qnet.algebra.core.algebraic_properties.**match_replace**(*cls*, *ops*, *kwargs*)

Match and replace a full operand specification to a function that provides a replacement for the whole expression or raises a *CannotSimplify* exception. E.g.

First define an operation:

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

Then some _rules:

```
>>> A = wc("A")
>>> A_float = wc("A", head=float)
>>> Invert_A = pattern(Invert, A)
>>> Invert._rules.update([
...     ('r1', (pattern_head(Invert_A), lambda A: A)),
...     ('r2', (pattern_head(A_float), lambda A: 1./A)),
... ])
```

Check rule application:

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

A pattern can also have the same wildcard appear twice:

```
>>> class X(Operation):
...     _rules = {
...         'r1': (pattern_head(A, A), lambda A: A),
...     }
...     simplifications = [match_replace, ]
>>> X.create(1,2)
X(1, 2)
>>> X.create(1,1)
1
```

`qnet.algebra.core.algebraic_properties.`**`match_replace_binary`**(*cls*, *ops*, *kwargs*)

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

```
>>> A = wc("A")
>>> class FilterDupes(Operation):
...     _binary_rules = {
...          'filter_dupes': (pattern_head(A,A), lambda A: A)}
...     simplifications = [match_replace_binary, assoc]
...     _neutral_element = 0
>>> FilterDupes.create(1,2,3,4)         # No duplicates
FilterDupes(1, 2, 3, 4)
>>> FilterDupes.create(1,2,2,3,4)       # Some duplicates
FilterDupes(1, 2, 3, 4)
```

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

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

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

`qnet.algebra.core.algebraic_properties.`**`check_cdims`**(*cls*, *ops*, *kwargs*)

Check that all operands (*ops*) have equal channel dimension.

`qnet.algebra.core.algebraic_properties.`**`filter_cid`**(*cls*, *ops*, *kwargs*)

Remove occurrences of the `circuit_identity()` `cid(n)` for any `n`. Cf. `filter_neutral()`

`qnet.algebra.core.algebraic_properties.`**`convert_to_spaces`**(*cls*, *ops*, *kwargs*)

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

`qnet.algebra.core.algebraic_properties.`**`empty_trivial`**(*cls*, *ops*, *kwargs*)

A ProductSpace of zero Hilbert spaces should yield the TrivialSpace

`qnet.algebra.core.algebraic_properties.`**`implied_local_space`**(*, *arg_index=None*, *keys=None*)

Return a simplification that converts the positional argument *arg_index* from (str, int) to a subclass of

`LocalSpace`, as well as any keyword argument with one of the given keys.

The exact type of the resulting Hilbert space is determined by the *default_hs_cls* argument of `init_algebra()`.

In many cases, we have `implied_local_space()` (in `create`) in addition to a conversion in `__init__`, so that `match_replace()` etc can rely on the relevant arguments being a `HilbertSpace` instance.

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

`qnet.algebra.core.algebraic_properties.`**`scalars_to_op`**`(`*cls*, *ops*, *kwargs*`)`
Convert any scalar α in *ops* into an operator $alpha identity$

`qnet.algebra.core.algebraic_properties.`**`convert_to_scalars`**`(`*cls*, *ops*, *kwargs*`)`
Convert any entry in *ops* that is not a `Scalar` instance into a `ScalarValue` instance

`qnet.algebra.core.algebraic_properties.`**`disjunct_hs_zero`**`(`*cls*, *ops*, *kwargs*`)`
Return ZeroOperator if all the operators in *ops* have a disjunct Hilbert space, or an unchanged *ops*, *kwargs* otherwise

`qnet.algebra.core.algebraic_properties.`**`commutator_order`**`(`*cls*, *ops*, *kwargs*`)`
Apply anti-commutative property of the commutator to apply a standard ordering of the commutator arguments

`qnet.algebra.core.algebraic_properties.`**`accept_bras`**`(`*cls*, *ops*, *kwargs*`)`
Accept operands that are all bras, and turn that into to bra of the operation applied to all corresponding kets

`qnet.algebra.core.algebraic_properties.`**`basis_ket_zero_outside_hs`**`(`*cls*, *ops*, *kwargs*`)`
For `BasisKet.create(ind, hs)` with an integer label *ind*, return a `ZeroKet` if *ind* is outside of the range of the underlying Hilbert space

`qnet.algebra.core.algebraic_properties.`**`indexed_sum_over_const`**`(`*cls*, *ops*, *kwargs*`)`
Execute an indexed sum over a term that does not depend on the summation indices

$$\sum_{j=1}^{N} a = Na$$

```
>>> a = symbols('a')
>>> i, j  = (IdxSym(s) for s in ('i', 'j'))
>>> unicode(Sum(i, 1, 2)(a))
'2 a'
>>> unicode(Sum(j, 1, 2)(Sum(i, 1, 2)(a * i)))
'_{i=1}^{2} 2 i a'
```

`qnet.algebra.core.algebraic_properties.`**`indexed_sum_over_kronecker`**`(`*cls*, *ops*, *kwargs*`)`
Execute sums over KroneckerDelta prefactors

`qnet.algebra.core.algebraic_properties.`**`derivative_via_diff`**`(`*cls*, *ops*, *kwargs*`)`
Implementation of the `QuantumDerivative.create()` interface via the use of `QuantumExpression._diff()`.

Thus, by having `QuantumExpression.diff()` delegate to `QuantumDerivative.create()`, instead of `QuantumExpression._diff()` directly, we get automatic caching of derivatives

qnet.algebra.core.circuit_algebra module

Implementation of the SLH circuit algebra

For more details see *Circuit Algebra*.

Summary

Classes:

`CPermutation`	Channel permuting circuit
`Circuit`	Base class for the circuit algebra elements
`CircuitSymbol`	Symbolic circuit element
`Component`	Base class for circuit components
`Concatenation`	Concatenation of circuit elements
`Feedback`	Feedback on a single channel of a circuit
`SLH`	Element of the SLH algebra
`SeriesInverse`	Symbolic series product inversion operation
`SeriesProduct`	The series product circuit operation.

Functions:

`FB`	Wrapper for `Feedback`, defaulting to last channel
`circuit_identity`	Return the circuit identity for n channels
`eval_adiabatic_limit`	Compute the limiting SLH model for the adiabatic approximation
`extract_channel`	Create a `CPermutation` that extracts channel *k*
`getABCD`	Calculate the ABCD-linearization of an SLH model
`map_channels`	Create a `CPermuation` based on a dict of channel mappings
`move_drive_to_H`	Move coherent drives from the Lindblad operators to the Hamiltonian.
`pad_with_identity`	Pad a circuit by adding a *n*-channel identity circuit at index *k*
`prepare_adiabatic_limit`	Prepare the adiabatic elimination on an SLH object
`try_adiabatic_elimination`	Attempt to automatically do adiabatic elimination on an SLH object

Data:

`CIdentity`	Single pass-through channel; neutral element of `SeriesProduct`
`CircuitZero`	Zero circuit, the neutral element of `Concatenation`

`__all__`: `CIdentity`, `CPermutation`, `Circuit`, `CircuitSymbol`, `CircuitZero`, `Component`, `Concatenation`, `FB`, `Feedback`, `SLH`, `SeriesInverse`, `SeriesProduct`, `circuit_identity`, `eval_adiabatic_limit`, `extract_channel`, `getABCD`, `map_channels`, `move_drive_to_H`, `pad_with_identity`, `prepare_adiabatic_limit`, `try_adiabatic_elimination`

Reference

class `qnet.algebra.core.circuit_algebra.`**`Circuit`**

Bases: `object`

Base class for the circuit algebra elements

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

Return type `int`

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

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

Then the block structure of their Concatenation is:

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

while

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

See also:

get_blocks() allows to actually retrieve the blocks:

```
>>> (A + B).get_blocks()
(A, B)
```

Return type `tuple`

index_in_block (*channel_index*)
Return the index a channel has within the subblock it belongs to

I.e., only for reducible circuits, this gives a result different from the argument itself.

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

Raises `ValueError` – for an invalid *channel_index*

Return type `int`

get_blocks (*block_structure=None*)
For a reducible circuit, get a sequence of subblocks that when concatenated again yield the original circuit. The block structure given has to be compatible with the circuits actual block structure, i.e. it can only be more coarse-grained.

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

Returns A tuple of subblocks that the circuit consists of.

Raises *IncompatibleBlockStructures*

series_inverse()

Return the inverse object (under the series product) for a circuit

In general for any X

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

Return type *Circuit*

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

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

Parameters

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

show()

Show the circuit expression in an IPython notebook.

render(*fname=''*)

Render the circuit expression and store the result in a file

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

Returns The path to the image file

Return type str

creduce()

If the circuit is reducible, try to reduce each subcomponent once

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

Return type *Circuit*

toSLH()

Return the SLH representation of a circuit. This can fail if there are un-substituted pure circuit symbols (*CircuitSymbol*) left in the expression

Return type *SLH*

coherent_input(**input_amps*)

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

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

Returns The circuit including the coherent inputs.

Return type *Circuit*

Raises *WrongCDimError*

class qnet.algebra.core.circuit_algebra.**SLH**(*S*, *L*, *H*)

Bases: *qnet.algebra.core.circuit_algebra.Circuit*, *qnet.algebra.core.abstract_algebra.Expression*

Element of the SLH algebra

The SLH class encapsulate an open system model that is parametrized the a scattering matrix (S), a column vector of Lindblad operators (L), and a Hamiltonian (H).

Parameters

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

S

Scattering matrix

L

Coupling vector

H

Hamiltonian

args

The tuple of positional arguments for the instantiation of the Expression

Ls

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

cdim

The circuit dimension

space

Total Hilbert space

free_symbols

Set of all symbols occcuring in S, L, or H

series_with_slh(*other*)

Series product with another *SLH* object

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

Returns The combined system.

Return type *SLH*

concatenate_slh(*other*)

Concatenation with another *SLH* object

expand()

Expand out all operator expressions within S, L and H

Return a new *SLH* object with these expanded expressions.

simplify_scalar(*func=<function simplify>*)

Simplify all scalar expressions within S, L and H

Return a new *SLH* object with the simplified expressions.

See also: *QuantumExpression.simplify_scalar()*

symbolic_liouvillian()

symbolic_master_equation(*rho=None*)

Compute the symbolic Liouvillian acting on a state rho

If no rho is given, an OperatorSymbol is created in its place. This correspnds to the RHS of the master equation in which an average is taken over the external noise degrees of freedom.

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

Returns The RHS of the master equation.

Return type *Operator*

symbolic_heisenberg_eom(*X=None*, *noises=None*, *expand_simplify=True*)

Compute the symbolic Heisenberg equations of motion of a system operator X. If no X is given, an OperatorSymbol is created in its place. If no noises are given, this correspnds to the ensemble-averaged Heisenberg equation of motion.

Parameters

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

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

Return type *Operator*

class qnet.algebra.core.circuit_algebra.**CircuitSymbol**(*label*, **sym_args*, *cdim*)

Bases: *qnet.algebra.core.circuit_algebra.Circuit*, *qnet.algebra.core.abstract_algebra.Expression*

Symbolic circuit element

Parameters

- **label** (*str*) – Label for the symbol
- **sym_args** (`Scalar`) – optional scalar arguments. With zero *sym_args*, the resulting symbol is a constant. With one or more *sym_args*, it becomes a function.
- **cdim** (*int*) – The circuit dimension, that is, the number of I/O lines

label

args

The tuple of positional arguments for the instantiation of the Expression

kwargs

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

sym_args

Tuple of arguments of the symbol

cdim

Dimension of circuit

class qnet.algebra.core.circuit_algebra.**Component**(**, label=None*, ***kwargs*)

Bases: *qnet.algebra.core.circuit_algebra.CircuitSymbol*

Base class for circuit components

A circuit component is a *CircuitSymbol* that knows its own SLH representation. Consequently, it has a fixed number of I/O channels (*CDIM* class attribute), and a fixed number of named arguments. Components only accept keyword arguments.

Any subclass of *Component* must define all of the class attributes listed below, and the `_toSLH()` method that return the *SLH* object for the component. Subclasses must also use the *properties_for_args()* class decorator:

```
@partial(properties_for_args, arg_names='ARGNAMES')
```

Parameters

- **label** (*str*) – label for the component. Defaults to *IDENTIFIER*
- **kwargs** – values for the parameters in *ARGNAMES*

Class Attributes

- **CDIM** – the circuit dimension (number of I/O channels)
- **PORTSIN** – list of names for the input ports of the component
- **PORTSOUT** – list of names for the output ports of the component
- **ARGNAMES** – the name of the keyword-arguments for the components (excluding `'label'`)
- **DEFAULTS** – mapping of keyword-argument names to default values
- **IDENTIFIER** – the default *label*

Note: The port names defined in *PORTSIN* and *PORTSOUT* may be used when defining connection via *connect()*.

See also:

qnet.algebra.library.circuit_components for example *Component* subclasses.

CDIM = 0

PORTSIN = ()

PORTSOUT = ()

ARGNAMES = ()

DEFAULTS = {}

IDENTIFIER = ''

args

Empty tuple (no arguments)

See also:

sym_args is a tuple of the keyword argument values.

kwargs

An `OrderedDict` with the value for the *label* argument, as well as any name in *ARGNAMES*

minimal_kwargs

An `OrderedDict` with the keyword arguments necessary to instantiate the component.

class qnet.algebra.core.circuit_algebra.**CPermutation**(*permutation*)

Bases: *qnet.algebra.core.circuit_algebra.Circuit*, *qnet.algebra.core.abstract_algebra.Expression*

Channel permuting circuit

This circuit expression is only a rearrangement of input and output fields. A channel permutation is given as a tuple of image points. A permutation $\sigma \in \Sigma_n$ of n elements is often represented in the following form

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

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

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

simplifications = []

classmethod create(*permutation*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

args

The tuple of positional arguments for the instantiation of the Expression

block_perms

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

Type tuple

permutation

The permutation image tuple.

cdim

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

series_with_permutation(*other*)

Compute the series product with another channel permutation circuit

Parameters **other** (`CPermutation`) –

Returns

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

Return type *Circuit*

qnet.algebra.core.circuit_algebra.**CIdentity = CIdentity**

Single pass-through channel; neutral element of `SeriesProduct`

qnet.algebra.core.circuit_algebra.**CircuitZero = CircuitZero**

Zero circuit, the neutral element of `Concatenation`

No ports, no internal dynamics.

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

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

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

simplifications = [<function assoc>, <function filter_cid>, <function check_cdims>, <f

neutral_element = CIdentity

Single pass-through channel; neutral element of *SeriesProduct*

cdim

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

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

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

Concatenation of circuit elements

simplifications = [<function assoc>, <function filter_neutral>, <function match_replac

neutral_element = CircuitZero

Zero circuit, the neutral element of *Concatenation*

No ports, no internal dynamics.

cdim

Circuit dimension (sum of dimensions of the operands)

class qnet.algebra.core.circuit_algebra.**Feedback**(*circuit*, ***, *out_port*, *in_port*)

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

Feedback on a single channel of a circuit

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

Parameters

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

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

simplifications = [<function match_replace>]

kwargs

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

operand

The Circuit that undergoes feedback

out_in_pair

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

cdim

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

classmethod create(*circuit*, ***, *out_port*, *in_port*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

Return type `Feedback`

class qnet.algebra.core.circuit_algebra.**SeriesInverse**(**operands*, ***kwargs*)

Bases: `qnet.algebra.core.circuit_algebra.Circuit`, `qnet.algebra.core.abstract_algebra.Operation`

Symbolic series product inversion operation

```
SeriesInverse(circuit)
```

One generally has

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

and

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

simplifications = []

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

operand

The un-inverted circuit

classmethod create(*circuit*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

cdim

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

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

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

Returns n-channel identity circuit

Return type *Circuit*

`qnet.algebra.core.circuit_algebra.`**`FB`**`(`*circuit*, *, *out_port=None*, *in_port=None*`)`
Wrapper for `Feedback`, defaulting to last channel

Parameters

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

Returns The circuit with applied feedback operation.

Return type *Circuit*

`qnet.algebra.core.circuit_algebra.`**`extract_channel`**`(`*k*, *cdim*`)`
Create a `CPermutation` that extracts channel *k*

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

Parameters

- **k** (*int*) – Extracted channel index
- **cdim** (*int*) – The circuit dimension (number of channels)

Returns Permutation circuit

Return type *Circuit*

`qnet.algebra.core.circuit_algebra.`**`map_channels`**`(`*mapping*, *cdim*`)`
Create a `CPermuation` based on a dict of channel mappings

For a given mapping in form of a dictionary, generate the channel permutating circuit that achieves the specified mapping while leaving the relative order of all non-specified channels intact.

Parameters

- **mapping** (*dict*) – Input-output mapping of indices (zero-based) `{in1:out1, in2:out2,...}`
- **cdim** (*int*) – The circuit dimension (number of channels)

Returns Circuit mapping the channels as specified

Return type *CPermutation*

`qnet.algebra.core.circuit_algebra.`**`pad_with_identity`**`(`*circuit*, *k*, *n*`)`
Pad a circuit by adding a *n*-channel identity circuit at index *k*

That is, a circuit of channel dimension N is extended to one of channel dimension $N + n$, where the channels k, $k + 1$, . . . $k+n-1$, just pass through the system unaffected. E.g. let `A`, `B` be two single channel systems:

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

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

Parameters

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

Returns

An extended circuit that passes through the channels $k, k+1, \ldots, k+n-1$

Return type *Circuit*

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

Calculate the ABCD-linearization of an SLH model

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

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

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

where, if doubled_up == False

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

or if doubled_up == True

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

Parameters

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

Returns

A tuple (A, B, C, D, a, c])

with

- *A*: coupling of modes to each other
- *B*: coupling of external input fields to modes
- *C*: coupling of internal modes to output
- *D*: coupling of external input fields to output fields
- *a*: constant coherent input vector for mode e.o.m.
- ***c*: constant coherent input vector of scattered amplitudes** contributing to the output

`qnet.algebra.core.circuit_algebra.`**`move_drive_to_H`**(*slh*, *which=None*, *expand_simplify=True*)

Move coherent drives from the Lindblad operators to the Hamiltonian.

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

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

$$L_i \longrightarrow L_i' = L_i - \alpha_i \tag{9.1}$$

$$H \longrightarrow H' = H + \frac{1}{2i} \sum_j (\alpha_j L_j^\dagger - \alpha_j^* L_j) \tag{9.2}$$

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

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

where α has the elements α_i.

Parameters

- **slh** (`SLH`) – SLH model to transform. If *slh* does not contain any inhomogeneities, it is invariant under the transformation.
- **which** (*sequence or None*) – Sequence of circuit dimensions to apply the transform to. If None, all dimensions are transformed.
- **expand_simplify** (*bool*) – if True, expand and simplify the new SLH object before returning. This has no effect if *slh* does not contain any inhomogeneities.

Returns **new_slh** – Transformed SLH model.

Return type *SLH*

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

Prepare the adiabatic elimination on an SLH object

Args: slh: The SLH object to take the limit for k: The scaling parameter $k

ightarrow infty$. The default is a

positive symbol 'k'

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

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

Compute the limiting SLH model for the adiabatic approximation

Parameters

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

Returns Limiting SLH model

Return type *SLH*

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

Attempt to automatically do adiabatic elimination on an SLH object

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

qnet.algebra.core.exceptions module

Exceptions and Errors raised by QNET

Summary

Exceptions:

AlgebraError	Base class for all algebraic errors
AlgebraException	Base class for all algebraic exceptions
BadLiouvillianError	Raised when a Liouvillian is not of standard Lindblad form.
BasisNotSetError	Raised if the basis or a Hilbert space dimension is unavailable
CannotConvertToSLH	Raised when a circuit algebra object cannot be converted to SLH
CannotEliminateAutomatically	Raised when attempted automatic adiabatic elimination fails.
CannotSimplify	Raised when a rule cannot further simplify an expression
CannotSymbolicallyDiagonalize	Matrix cannot be diagonalized analytically.
CannotVisualize	Raised when a circuit cannot be visually represented.
IncompatibleBlockStructures	Raised for invalid block-decomposition
InfiniteSumError	Raised when expanding a sum into an infinite number of terms
NoConjugateMatrix	Raised when entries of *Matrix* have no defined conjugate
NonSquareMatrix	Raised when a *Matrix* fails to be square
OverlappingSpaces	Raised when objects fail to be in separate Hilbert spaces.
SpaceTooLargeError	Raised when objects fail to be have overlapping Hilbert spaces.
UnequalSpaces	Raised when objects fail to be in the same Hilbert space.
WrongCDimError	Raised for mismatched channel number in circuit series

`__all__`: *AlgebraError*, *AlgebraException*, *BadLiouvillianError*, *BasisNotSetError*, *CannotConvertToSLH*, *CannotEliminateAutomatically*, *CannotSimplify*, *CannotSymbolicallyDiagonalize*, *CannotVisualize*, *IncompatibleBlockStructures*, *InfiniteSumError*, *NoConjugateMatrix*, *NonSquareMatrix*, *OverlappingSpaces*, *SpaceTooLargeError*, *UnequalSpaces*, *WrongCDimError*

Reference

exception qnet.algebra.core.exceptions.**AlgebraException**

Bases: `Exception`

Base class for all algebraic exceptions

exception qnet.algebra.core.exceptions.**AlgebraError**

Bases: *qnet.algebra.core.exceptions.AlgebraException*

Base class for all algebraic errors

exception qnet.algebra.core.exceptions.**InfiniteSumError**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised when expanding a sum into an infinite number of terms

exception qnet.algebra.core.exceptions.**CannotSimplify**
Bases: *qnet.algebra.core.exceptions.AlgebraException*

Raised when a rule cannot further simplify an expression

exception qnet.algebra.core.exceptions.**CannotConvertToSLH**
Bases: *qnet.algebra.core.exceptions.AlgebraException*

Raised when a circuit algebra object cannot be converted to SLH

exception qnet.algebra.core.exceptions.**CannotVisualize**
Bases: *qnet.algebra.core.exceptions.AlgebraException*

Raised when a circuit cannot be visually represented.

exception qnet.algebra.core.exceptions.**WrongCDimError**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised for mismatched channel number in circuit series

exception qnet.algebra.core.exceptions.**IncompatibleBlockStructures**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised for invalid block-decomposition

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

exception qnet.algebra.core.exceptions.**CannotEliminateAutomatically**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised when attempted automatic adiabatic elimination fails.

exception qnet.algebra.core.exceptions.**BasisNotSetError**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised if the basis or a Hilbert space dimension is unavailable

exception qnet.algebra.core.exceptions.**UnequalSpaces**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised when objects fail to be in the same Hilbert space.

This happens for example when trying to add two states from different Hilbert spaces.

exception qnet.algebra.core.exceptions.**OverlappingSpaces**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised when objects fail to be in separate Hilbert spaces.

exception qnet.algebra.core.exceptions.**SpaceTooLargeError**
Bases: *qnet.algebra.core.exceptions.AlgebraError*

Raised when objects fail to be have overlapping Hilbert spaces.

exception qnet.algebra.core.exceptions.**CannotSymbolicallyDiagonalize**
Bases: *qnet.algebra.core.exceptions.AlgebraException*

Matrix cannot be diagonalized analytically.

Signals that a fallback to numerical diagonalization is required.

exception `qnet.algebra.core.exceptions.`**`BadLiouvillianError`**
Bases: *`qnet.algebra.core.exceptions.AlgebraError`*

Raised when a Liouvillian is not of standard Lindblad form.

exception `qnet.algebra.core.exceptions.`**`NonSquareMatrix`**
Bases: *`qnet.algebra.core.exceptions.AlgebraError`*

Raised when a *`Matrix`* fails to be square

exception `qnet.algebra.core.exceptions.`**`NoConjugateMatrix`**
Bases: *`qnet.algebra.core.exceptions.AlgebraError`*

Raised when entries of *`Matrix`* have no defined conjugate

qnet.algebra.core.hilbert_space_algebra module

Core class hierarchy for Hilbert spaces

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 *Algebraic Manipulations*.

Summary

Classes:

`HilbertSpace`	Base class for Hilbert spaces
`LocalSpace`	Hilbert space for a single degree of freedom.
`ProductSpace`	Tensor product of local Hilbert spaces

Data:

`FullSpace`	The 'full space', i.e.
`TrivialSpace`	The 'nullspace', i.e.

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

Reference

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

Base class for Hilbert spaces

`tensor`(**others*)
Tensor product between Hilbert spaces

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

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

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

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

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

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

Return type bool

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

dimension
Full dimension of the Hilbert space.

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

has_basis
True if the Hilbert space has a basis

basis_states
Yield an iterator over the states (*State* instances) that form the canonical basis of the Hilbert space

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

basis_state(*index_or_label*)
Return the basis state with the given index or label.

Raises

- *BasisNotSetError* – if the Hilbert space has no defined basis
- `IndexError` – if there is no basis state with the given index
- `KeyError` – if there is not basis state with the given label

basis_labels
Tuple of basis labels.

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

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

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

class qnet.algebra.core.hilbert_space_algebra.**LocalSpace**(*label*, *, *basis=None*, *dimension=None*, *local_identifiers=None*, *order_index=None*)

Bases: *qnet.algebra.core.hilbert_space_algebra.HilbertSpace*, *qnet.algebra.core.abstract_algebra.Expression*

Hilbert space for a single degree of freedom.

Parameters

- **label** (*str or int or* `StrLabel`) – label (subscript) of the Hilbert space

- **basis** (*tuple or None*) – Set an explicit basis for the Hilbert space (tuple of labels for the basis states)
- **dimension** (*int or None*) – Specify the dimension n of the Hilbert space. This implies a basis numbered from 0 to $n - 1$.
- **local_identifiers** (*dict*) – Mapping of class names of *LocalOperator* subclasses to identifier names. Used e.g. 'b' instead of the default 'a' for the anihilation operator. This can be a dict or a dict-compatible structure, e.g. a list/tuple of key-value tuples.
- **order_index** (*int or None*) – An optional key that determines the preferred order of Hilbert spaces. This also changes the order of e.g. sums or products of Operators. Hilbert spaces will be ordered from left to right be increasing *order_index*; Hilbert spaces without an explicit *order_index* are sorted by their label

A *LocalSpace* fundamentally has a Fock-space like structure, in that its basis states may be understood as an "excitation". The spectrum can be infinite, with levels labeled by integers 0, 1, . . . :

```
>>> hs = LocalSpace(label=0)
```

or truncated to a finite dimension:

```
>>> hs = LocalSpace(0, dimension=5)
>>> hs.basis_labels
('0', '1', '2', '3', '4')
```

For finite-dimensional (truncated) Hilbert spaces, we also allow an arbitrary alternative labeling of the canonical basis:

```
>>> hs = LocalSpace('rydberg', dimension=3, basis=('g', 'e', 'r'))
```

args
List of arguments, consisting only of *label*

label
Label of the Hilbert space

has_basis
True if the Hilbert space has a basis

basis_states
Yield an iterator over the states (*BasisKet* instances) that form the canonical basis of the Hilbert space

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

basis_state (*index_or_label*)
Return the basis state with the given index or label.

Raises

- *BasisNotSetError* – if the Hilbert space has no defined basis
- IndexError – if there is no basis state with the given index
- KeyError – if there is not basis state with the given label

basis_labels
Tuple of basis labels (strings).

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

dimension
Dimension of the Hilbert space.

Raises *BasisNotSetError* – if the Hilbert space has no defined basis

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

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

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

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

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

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

next_basis_label_or_index(*label_or_index*, *n=1*)
Given the label or index of a basis state, return the label/index of the next basis state.

More generally, if *n* is given, return the *n*'th next basis state label/index; *n* may also be negative to obtain previous basis state labels/indices.

The return type is the same as the type of *label_or_index*.

Parameters

- **label_or_index** (*int or str or* SymbolicLabelBase) – If *int*, the index of a basis state; if *str*, the label of a basis state
- **n** (*int*) – The increment

Raises

- IndexError – If going beyond the last or first basis state
- ValueError – If *label* is not a label for any basis state in the Hilbert space
- *BasisNotSetError* – If the Hilbert space has no defined basis
- TypeError – if *label_or_index* is neither a str nor an int, nor a SymbolicLabelBase

qnet.algebra.core.hilbert_space_algebra.**TrivialSpace = TrivialSpace**
The 'nullspace', i.e. a one dimensional Hilbert space, which is a factor space of every other Hilbert space.

This is the Hilbert space of scalars.

qnet.algebra.core.hilbert_space_algebra.**FullSpace = FullSpace**
The 'full space', i.e. a Hilbert space that includes any other Hilbert space as a tensor factor.

The *FullSpace* has no defined basis, any related properties will raise *BasisNotSetError*

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

Tensor product of local Hilbert spaces

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

simplifications = [<function empty_trivial>, <function assoc>, <function convert_to_spa

classmethod create(**local_spaces*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

has_basis

True if the all the local factors of the *ProductSpace* have a defined basis

basis_states

Yield an iterator over the states (`TensorKet` instances) that form the canonical basis of the Hilbert space

Raises `BasisNotSetError` – if the Hilbert space has no defined basis

basis_labels

Tuple of basis labels. Each basis label consists of the labels of the `BasisKet` states that factor the basis state, separated by commas.

Raises `BasisNotSetError` – if the Hilbert space has no defined basis

basis_state(*index_or_label*)

Return the basis state with the given index or label.

Raises

- `BasisNotSetError` – if the Hilbert space has no defined basis
- `IndexError` – if there is no basis state with the given index
- `KeyError` – if there is not basis state with the given label

dimension

Dimension of the Hilbert space.

Raises `BasisNotSetError` – if the Hilbert space has no defined basis

remove(*other*)

Remove a particular factor from a tensor product space.

local_factors

The `LocalSpace` instances that make up the product

classmethod order_key(*obj*)

Key by which operands are sorted

intersect(*other*)

Find the mutual tensor factors of two Hilbert spaces.

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

qnet.algebra.core.indexed_operations module

Base classes for indexed operations (sums and products)

Summary

Classes:

IndexedSum	Base class for indexed sums

`__all__`: *IndexedSum*

Reference

class qnet.algebra.core.indexed_operations.**IndexedSum**(*term*, **ranges*)
Bases: *qnet.algebra.core.abstract_algebra.Operation*

Base class for indexed sums

term

operands
Tuple of operands of the operation

args
Alias for operands

variables
List of the dummy (index) variable symbols

See also **:property:'bound_symbols'** for a set of the same symbols

bound_symbols
Set of bound variables, i.e. the index variable symbols

See also **:property:'variables'** for an ordered list of the same symbols

free_symbols
Set of all free symbols

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

terms
Iterator over the terms of the sum

Yield from the (possibly) infinite list of terms of the indexed sum, if the sum was written out explicitly. Each yielded term in an instance of *Expression*

doit(*classes=None*, *recursive=True*, *indices=None*, *max_terms=None*, ***kwargs*)
Write out the indexed sum explicitly

If *classes* is None or *IndexedSum* is in *classes*, (partially) write out the indexed sum in to an explicit sum of terms. If *recursive* is True, write out each of the new sum's summands by calling its *doit()* method.

Parameters

- **classes** (*None or list*) – see *Expression.doit()*
- **recursive** (*bool*) – see *Expression.doit()*
- **indices** (*list*) – List of `IdxSym` indices for which the sum should be expanded. If *indices* is a subset of the indices over which the sum runs, it will be partially expanded. If not given, expand the sum completely
- **max_terms** (*int*) – Number of terms after which to truncate the sum. This is particularly useful for infinite sums. If not given, expand all terms of the sum. Cannot be combined with *indices*
- **kwargs** – keyword arguments for recursive calls to *doit()*. See *Expression.doit()*

make_disjunct_indices(**others*)
Return a copy with modified indices to ensure disjunct indices with *others*.

Each element in *others* may be an index symbol (*IdxSym*), a index-range object (*IndexRangeBase*) or list of index-range objects, or an indexed operation (something with a `ranges` attribute)

Each index symbol is primed until it does not match any index symbol in *others*.

qnet.algebra.core.matrix_algebra module

Matrices of Operators

Summary

Classes:

Matrix	Matrix of Expressions

Functions:

block_matrix	Generate the operator matrix with quadrants
diagm	Generalizes the diagonal matrix creation capabilities of *numpy.diag* to *Matrix* objects.
hstackm	Generalizes *numpy.hstack* to *Matrix* objects.
identity_matrix	Generate the N-dimensional identity matrix.
permutation_matrix	Return orthogonal permutation matrix for permutation tuple
vstackm	Generalizes *numpy.vstack* to *Matrix* objects.
zerosm	Generalizes `numpy.zeros` to *Matrix* objects.

`__all__`: *Matrix*, *block_matrix*, *diagm*, *hstackm*, *identity_matrix*, *vstackm*, *zerosm*

Reference

class qnet.algebra.core.matrix_algebra.**Matrix**(*m*)
Bases: *qnet.algebra.core.abstract_algebra.Expression*

Matrix of Expressions

Matrices of `Operator` expressions are required for the SLH formalism.

matrix = None

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

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

Return type tuple

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

is_zero
Are all elements of the matrix zero?

transpose()
The transpose matrix

conjugate()
The element-wise conjugate matrix

This is defined only if all the entries in the matrix have a defined conjugate (i.e., they have a *conjugate* method). This is *not* the case for a matrix of operators. In such a case, only an `elementwise()` `adjoint()` would be applicable, but this is mathematically different from a complex conjugate.

Raises `NoConjugateMatrix` – if any entries have no *conjugate* method

real
Element-wise real part

Raises `NoConjugateMatrix` – if entries have no *conjugate* method and no other way to determine the real part

Note: A mathematically equivalent way to obtain a real matrix from a complex matrix `M` is:

```
(M.conjugate() + M) / 2
```

However, the result may not be identical to `M.real`, as the latter tries to convert elements of the matrix to real values directly, if possible, and only uses the conjugate as a fall-back

imag
Element-wise imaginary part

Raises `NoConjugateMatrix` – if entries have no *conjugate* method and no other way to determine the imaginary part

Note: A mathematically equivalent way to obtain an imaginary matrix from a complex matrix `M` is:

```
(M.conjugate() - M) / (I * 2)
```

with same same caveats as `real`.

T
Alias for `transpose()`

adjoint()
Adjoint of the matrix

This is the transpose and the Hermitian adjoint of all elements.

dag()
Adjoint of the matrix

This is the transpose and the Hermitian adjoint of all elements.

trace()

H
Alias for `adjoint()`

element_wise(*func*, **args*, ***kwargs*)
Apply a function to each matrix element and return the result in a new operator matrix of the same shape.

Parameters

- **func** (*FunctionType*) – A function to be applied to each element. It must take the element as its first argument.
- **args** – Additional positional arguments to be passed to *func*
- **kwargs** – Additional keyword arguments to be passed to *func*

Returns Matrix with results of *func*, applied element-wise.

Return type *Matrix*

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

Parameters

- **param** (Symbol) – Expansion parameter.
- **about** (Scalar) – Point about which to expand.
- **order** (int) – Maximum order of expansion >= 0

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

expand()
Expand each matrix element distributively.

Returns Expanded matrix.

Return type *Matrix*

free_symbols
Set of free SymPy symbols contained within the expression.

space
Combined Hilbert space of all matrix elements.

simplify_scalar(*func=<function simplify>*)
Simplify all scalar expressions appearing in the Matrix.

qnet.algebra.core.matrix_algebra.**hstackm**(*matrices*)
Generalizes *numpy.hstack* to `Matrix` objects.

qnet.algebra.core.matrix_algebra.**vstackm**(*matrices*)
Generalizes *numpy.vstack* to `Matrix` objects.

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

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

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

Parameters

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

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

Return type *Matrix*

`qnet.algebra.core.matrix_algebra.`**`identity_matrix`**(*N*)
Generate the N-dimensional identity matrix.

Parameters **N** (*int*) – Dimension

Returns Identity matrix in N dimensions

Return type *Matrix*

`qnet.algebra.core.matrix_algebra.`**`zerosm`**(*shape*, **args*, ***kwargs*)
Generalizes `numpy.zeros` to `Matrix` objects.

`qnet.algebra.core.matrix_algebra.`**`permutation_matrix`**(*permutation*)
Return orthogonal permutation matrix for permutation tuple

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

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

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

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

The column form of M_σ is thus given by

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

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

qnet.algebra.core.operator_algebra module

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

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

Summary

Classes:

Adjoint	Symbolic Adjoint of an operator
Commutator	Commutator of two operators
LocalOperator	Base class for "known" operators on a LocalSpace
LocalSigma	Level flip operator between two levels of a LocalSpace
NullSpaceProjector	Projection operator onto the nullspace of its operand
Operator	Base class for all quantum operators.
OperatorDerivative	Symbolic partial derivative of an operator
OperatorIndexedSum	Indexed sum over operators
OperatorPlus	Sum of Operators
OperatorPlusMinusCC	An operator plus or minus its complex conjugate
OperatorSymbol	Symbolic operator
OperatorTimes	Product of operators
OperatorTrace	(Partial) trace of an operator
PseudoInverse	Unevaluated pseudo-inverse X^+ of an operator X
ScalarTimesOperator	Product of a Scalar coefficient and an Operator

Functions:

LocalProjector	A projector onto a specific level of a LocalSpace
adjoint	Return the adjoint of an obj.
decompose_space	Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated partial traces.
factor_coeff	Factor out coefficients of all factors.
factor_for_trace	Given a LocalSpace *ls* to take the partial trace over and an operator *op*, factor the trace such that operators acting on disjoint degrees of freedom are pulled out of the trace.
get_coeffs	Create a dictionary with all Operator terms of the expression (understood as a sum) as keys and their coefficients as values.
rewrite_with_operator_pm_cc	Try to rewrite expr using OperatorPlusMinusCC

Data:

II	IdentityOperator constant (singleton) object.
IdentityOperator	IdentityOperator constant (singleton) object.
ZeroOperator	ZeroOperator constant (singleton) object.

`__all__`: Adjoint, Commutator, II, IdentityOperator, LocalOperator, LocalProjector, LocalSigma, NullSpaceProjector, Operator, OperatorDerivative, OperatorIndexedSum, OperatorPlus, OperatorPlusMinusCC, OperatorSymbol, OperatorTimes, OperatorTrace, PseudoInverse, ScalarTimesOperator, ZeroOperator, adjoint, decompose_space, factor_coeff, factor_for_trace, get_coeffs, rewrite_with_operator_pm_cc, tr

Reference

class qnet.algebra.core.operator_algebra.**Operator**(**args*, ***kwargs*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumExpression*

Base class for all quantum operators.

pseudo_inverse()

Pseudo-inverse X^+ of the operator X

It is defined via the relationship

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

expand_in_basis(*basis_states=None*, *hermitian=False*)

Write the operator as an expansion into all *KetBras* spanned by *basis_states*.

Parameters

- **basis_states** (*list or None*) – List of basis states (*State* instances) into which to expand the operator. If None, use the operator's *space.basis_states*
- **hermitian** (*bool*) – If True, assume that the operator is Hermitian and represent all elements in the lower triangle of the expansion via *OperatorPlusMinusCC*. This is meant to enhance readability

Raises *BasisNotSetError* – If *basis_states* is None and the operator's Hilbert space has no well-defined basis

Example

```
>>> hs = LocalSpace(1, basis=('g', 'e'))
>>> op = LocalSigma('g', 'e', hs=hs) + LocalSigma('e', 'g', hs=hs)
>>> print(ascii(op, sig_as_ketbra=False))
sigma_e,g^(1) + sigma_g,e^(1)
>>> print(ascii(op.expand_in_basis()))
|e><g|^(1) + |g><e|^(1)
>>> print(ascii(op.expand_in_basis(hermitian=True)))
|g><e|^(1) + c.c.
```

class qnet.algebra.core.operator_algebra.**LocalOperator**(**args*, *hs*)

Bases: *qnet.algebra.core.operator_algebra.Operator*

Base class for "known" operators on a `LocalSpace`

All *LocalOperator* instances have known algebraic properties and a fixed associated identifier (symbol) that is used when printing that operator. A custom identifier can be used through the associated *LocalSpace*'s *local_identifiers* parameter. For example:

```
>>> hs1_custom = LocalSpace(1, local_identifiers={'Destroy': 'b'})
>>> b = Destroy(hs=hs1_custom)
>>> ascii(b)
'b^(1)'
```

Note: It is recommended that subclasses use the *properties_for_args()* class decorator if they define any position arguments (via the `_arg_names` class attribute)

simplifications = [<function implied_local_space.<locals>.kwargs_to_local_space>]

space
Hilbert space of the operator (*LocalSpace* instance)

args
The positional arguments used for instantiating the operator

kwargs
The keyword arguments used for instantiating the operator

identifier
The identifier (symbol) that is used when printing the operator.

A custom identifier can be used through the associated *LocalSpace*'s *local_identifiers* parameter. For example:

```
>>> a = Destroy(hs=1)
>>> a.identifier
'a'
>>> hs1_custom = LocalSpace(1, local_identifiers={'Destroy': 'b'})
>>> b = Destroy(hs=hs1_custom)
>>> b.identifier
'b'
>>> ascii(b)
'b^(1)'
```

class qnet.algebra.core.operator_algebra.**OperatorSymbol**(*label*, **sym_args*, *hs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumSymbol*, *qnet.algebra.core.operator_algebra.Operator*

Symbolic operator

See *QuantumSymbol*.

qnet.algebra.core.operator_algebra.**IdentityOperator = IdentityOperator**
IdentityOperator constant (singleton) object.

qnet.algebra.core.operator_algebra.**II = IdentityOperator**
IdentityOperator constant (singleton) object.

qnet.algebra.core.operator_algebra.**ZeroOperator = ZeroOperator**
ZeroOperator constant (singleton) object.

class qnet.algebra.core.operator_algebra.**LocalSigma**(*j*, *k*, *, *hs*)
Bases: *qnet.algebra.core.operator_algebra.LocalOperator*

Level flip operator between two levels of a *LocalSpace*

$$\sigma_{jk}^{\rm hs} = \left\lvert j\right\rangle_{\rm hs} \left \langle k\right\rvert_{\rm hs}$$

For $j = k$ this becomes a projector P_k onto the eigenstate k; see *LocalProjector*.

Parameters

- **j** (*int or str*) – The label or index identifying j
- **k** (*int or str*) – The label or index identifying k
- **hs** (LocalSpace *or int or str*) – The Hilbert space on which the operator acts. If an int or a str, an implicit Hilbert space will be constructed as a subclass of LocalSpace, as configured by init_algebra().

Note: The parameters *j* or *k* may be an integer or a string. A string refers to the label of an eigenstate in the basis of *hs*, which needs to be set. An integer refers to the (zero-based) index of eigenstate of the Hilbert space. This works if *hs* has an unknown dimension. Assuming the Hilbert space has a defined basis, using integer or string labels is equivalent:

```
>>> hs = LocalSpace('tls', basis=('g', 'e'))
>>> LocalSigma(0, 1, hs=hs) == LocalSigma('g', 'e', hs=hs)
True
```

Raises `ValueError` – If *j* or *k* are invalid value for the given *hs*

Printers should represent this operator either in braket notation, or using the operator identifier

```
>>> LocalSigma(0, 1, hs=0).identifier
'sigma'
```

For `j == k`, an alternative (fixed) identifier may be used

```
>>> LocalSigma(0, 0, hs=0)._identifier_projector
'Pi'
```

simplifications = [<function implied_local_space.<locals>.kwargs_to_local_space>, <fun

args
The two eigenstate labels *j* and *k* that the operator connects

index_j
Index *j* or (zero-based) index of the label *j* in the basis

index_k
Index *k* or (zero-based) index of the label *k* in the basis

raise_jk (*j_incr=0*, *k_incr=0*)
Return a new *LocalSigma* instance with incremented *j*, *k*, on the same Hilbert space:

$$\sigma_{jk}^{\text{hs}} \rightarrow \sigma_{j'k'}^{\text{hs}}$$

This is the result of multiplying σ_{jk}^{hs} with any raising or lowering operators.

If j' or k' are outside the Hilbert space hs, the result is the *ZeroOperator* .

Parameters

- **j_incr** (*int*) – The increment between labels j and j'
- **k_incr** (*int*) – The increment between labels k and k'. Both increments may be negative.

j
The *j* argument.

k
The *k* argument.

qnet.algebra.core.operator_algebra.**LocalProjector** (*j*, ***, *hs*)
A projector onto a specific level of a *LocalSpace*

Parameters

- **j** (*int or str*) – The label or index identifying the state onto which is projected
- **hs** (HilbertSpace) – The Hilbert space on which the operator acts

class qnet.algebra.core.operator_algebra.**OperatorPlus**(**operands*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumPlus*, *qnet.algebra.core.operator_algebra.Operator*

Sum of Operators

simplifications = [<function assoc>, <function scalars_to_op>, <function orderby>, <fu

class qnet.algebra.core.operator_algebra.**OperatorTimes**(**operands*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumTimes*, *qnet.algebra.core.operator_algebra.Operator*

Product of operators

This serves both as a product within a Hilbert space as well as a tensor product.

simplifications = [<function assoc>, <function orderby>, <function filter_neutral>, <f

class qnet.algebra.core.operator_algebra.**ScalarTimesOperator**(*coeff*, *term*)
Bases: *qnet.algebra.core.operator_algebra.Operator*, *qnet.algebra.core.abstract_quantum_algebra.ScalarTimesQuantumExpression*

Product of a *Scalar* coefficient and an *Operator*

simplifications = [<function match_replace>]

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

class qnet.algebra.core.operator_algebra.**OperatorDerivative**(*op*, ***, *derivs*, *vals=None*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumDerivative*, *qnet.algebra.core.operator_algebra.Operator*

Symbolic partial derivative of an operator

See *QuantumDerivative*.

class qnet.algebra.core.operator_algebra.**Commutator**(*A*, *B*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumOperation*, *qnet.algebra.core.operator_algebra.Operator*

Commutator of two operators

$$[A, B] = AB - AB$$

simplifications = [<function scalars_to_op>, <function disjunct_hs_zero>, <function co

order_key
alias of *qnet.utils.ordering.FullCommutativeHSOrder*

A
Left side of the commutator

B
Left side of the commutator

doit(*classes=None*, *recursive=True*, ***kwargs*)
Write out commutator

Write out the commutator according to its definition $[A, B] = AB - AB$.

See *Expression.doit()*.

class qnet.algebra.core.operator_algebra.**OperatorTrace**(*op*, *, *over_space*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*, *qnet.algebra.core.operator_algebra.Operator*

(Partial) trace of an operator

Trace of an operator *op* ($Op{O}) over the degrees of freedom of a Hilbert space *over_space* ($mathcal{H}$):

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

Parameters

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

simplifications = [<function scalars_to_op>, <function implied_local_space.<locals>.kw

kwargs

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

operand

The operator that the operation acts on

space

Hilbert space of the operation result

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

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumAdjoint*, *qnet.algebra.core.operator_algebra.Operator*

Symbolic Adjoint of an operator

simplifications = [<function scalars_to_op>, <function delegate_to_method.<locals>._de

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

Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*, *qnet.algebra.core.operator_algebra.Operator*

An operator plus or minus its complex conjugate

kwargs

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

minimal_kwargs

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

doit(*classes=None*, *recursive=True*, ***kwargs*)

Write out the complex conjugate summand

See *Expression.doit()*.

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

Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*, *qnet.algebra.core.operator_algebra.Operator*

Unevaluated pseudo-inverse X^+ of an operator X

It is defined via the relationship

$$
\begin{aligned}
XX^{+}X &= X \\
X^{+}XX^{+} &= X^{+} \\
(X^{+}X)^{\dagger} &= X^{+}X \\
(XX^{+})^{\dagger} &= XX^{+}
\end{aligned}
$$

simplifications = [<function scalars_to_op>, <function delegate_to_method.<locals>._de

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

Bases: *qnet.algebra.core.abstract_quantum_algebra.SingleQuantumOperation*, *qnet.algebra.core.operator_algebra.Operator*

Projection operator onto the nullspace of its operand

Returns the operator $\mathcal{P}_{\mathrm{Ker}X}$ with

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

simplifications = [<function scalars_to_op>, <function match_replace>]

class qnet.algebra.core.operator_algebra.**OperatorIndexedSum**(*term*, **ranges*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumIndexedSum*, *qnet.algebra.core.operator_algebra.Operator*

Indexed sum over operators

simplifications = [<function assoc_indexed>, <function scalars_to_op>, <function index

qnet.algebra.core.operator_algebra.**factor_for_trace**(*ls*, *op*)

Given a *LocalSpace* *ls* to take the partial trace over and an operator *op*, factor the trace such that operators acting on disjoint degrees of freedom are pulled out of the trace. If the operator acts trivially on ls the trace yields only a pre-factor equal to the dimension of ls. If there are *LocalSigma* operators among a product, the trace's cyclical property is used to move to sandwich the full product by *LocalSigma* operators:

$$
\mathrm{Tr} A \sigma_{jk} B = \mathrm{Tr} \sigma_{jk} B A \sigma_{jj}
$$

Parameters

- **ls** (*HilbertSpace*) – Degree of Freedom to trace over
- **op** (*Operator*) – Operator to take the trace of

Return type *Operator*

Returns The (partial) trace over the operator's spc-degrees of freedom

qnet.algebra.core.operator_algebra.**decompose_space**(*H*, *A*)

Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated partial traces.

Parameters

- **H** (ProductSpace) – The full space.
- **A** (Operator) –

Returns Iterative partial trace expression

Return type *Operator*

`qnet.algebra.core.operator_algebra.`**`get_coeffs`**(*expr*, *expand=False*, *epsilon=0.0*)

Create a dictionary with all Operator terms of the expression (understood as a sum) as keys and their coefficients as values.

The returned object is a defaultdict that return 0. if a term/key doesn't exist.

Parameters

- **`expr`** – The operator expression to get all coefficients from.
- **`expand`** – Whether to expand the expression distributively.
- **`epsilon`** – If non-zero, drop all Operators with coefficients that have absolute value less than epsilon.

Returns A dictionary `{op1: coeff1, op2: coeff2, ...}`

Return type dict

`qnet.algebra.core.operator_algebra.`**`factor_coeff`**(*cls*, *ops*, *kwargs*)

Factor out coefficients of all factors.

`qnet.algebra.core.operator_algebra.`**`adjoint`**(*obj*)

Return the adjoint of an obj.

`qnet.algebra.core.operator_algebra.`**`rewrite_with_operator_pm_cc`**(*expr*)

Try to rewrite expr using `OperatorPlusMinusCC`

Example

```
>>> A = OperatorSymbol('A', hs=1)
>>> sum = A + A.dag()
>>> sum2 = rewrite_with_operator_pm_cc(sum)
>>> print(ascii(sum2))
A^(1) + c.c.
```

qnet.algebra.core.scalar_algebra module

Implementation of the scalar (quantum) algebra

Summary

Classes:

`Scalar`	Base class for Scalars
`ScalarDerivative`	Symbolic partial derivative of a scalar
`ScalarExpression`	Base class for scalars with non-scalar arguments
`ScalarIndexedSum`	Indexed sum over scalars
`ScalarPlus`	Sum of scalars
`ScalarPower`	A scalar raised to a power
`ScalarTimes`	Product of scalars
`ScalarValue`	Wrapper around a numeric or symbolic value

Functions:

`KroneckerDelta`	Kronecker delta symbol
`is_scalar`	Check if *scalar* is a `Scalar` or a scalar value
`sqrt`	Square root of a `Scalar` or scalar value

Data:

`One`	The neutral element with respect to scalar multiplication
`Zero`	The neutral element with respect to scalar addition

`__all__`: `KroneckerDelta`, `One`, `Scalar`, `ScalarDerivative`, `ScalarExpression`, `ScalarIndexedSum`, `ScalarPlus`, `ScalarPower`, `ScalarTimes`, `ScalarValue`, `Zero`, `sqrt`

Reference

class qnet.algebra.core.scalar_algebra.**Scalar**(**args*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumExpression`

Base class for Scalars

space

`TrivialSpace`, by definition

conjugate()

Complex conjugate

real

Real part

imag

Imaginary part

class qnet.algebra.core.scalar_algebra.**ScalarValue**(*val*)

Bases: `qnet.algebra.core.scalar_algebra.Scalar`

Wrapper around a numeric or symbolic value

The wrapped value may be of any of the following types:

```
>>> for t in ScalarValue._val_types:
...     print(t)
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'sympy.core.basic.Basic'>
<class 'numpy.int64'>
<class 'numpy.complex128'>
<class 'numpy.float64'>
```

A `ScalarValue` behaves exactly like its wrapped value in all algebraic contexts:

```
>>> 5 * ScalarValue.create(2)
10
```

Any unknown attributes or methods will be forwarded to the wrapped value to ensure complete “duck-typing”:

```
>>> alpha = ScalarValue(sympy.symbols('alpha', positive=True))
>>> alpha.is_positive   # same as alpha.val.is_positive
True
>>> ScalarValue(5).is_positive
Traceback (most recent call last):
  ...
AttributeError: 'int' object has no attribute 'is_positive'
```

classmethod create (*val*)

Instatiate the `ScalarValue` while recognizing `Zero` and `One`.

`Scalar` instances as *val* (including `ScalarExpression` instances) are left unchanged. This makes `ScalarValue.create()` a safe method for converting unknown objects to `Scalar`.

val

The wrapped scalar value

args

Tuple containing the wrapped scalar value as its only element

real

Real part

imag

Imaginary part

class qnet.algebra.core.scalar_algebra.**ScalarExpression** (**args*, ***kwargs*)

Bases: `qnet.algebra.core.scalar_algebra.Scalar`

Base class for scalars with non-scalar arguments

For example, a `BraKet` is a `Scalar`, but has arguments that are states.

qnet.algebra.core.scalar_algebra.**Zero = Zero**

The neutral element with respect to scalar addition

Equivalent to the scalar value zero:

```
>>> Zero == 0
True
```

qnet.algebra.core.scalar_algebra.**One = One**

The neutral element with respect to scalar multiplication

Equivalent to the scalar value one:

```
>>> One == 1
True
```

class qnet.algebra.core.scalar_algebra.**ScalarPlus** (**operands*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumPlus`, `qnet.algebra.core.scalar_algebra.Scalar`

Sum of scalars

Generally, `ScalarValue` instances are combined directly:

```
>>> alpha = ScalarValue.create(sympy.symbols('alpha'))
>>> print(srepr(alpha + 1))
ScalarValue(Add(Symbol('alpha'), Integer(1)))
```

An unevaluated `ScalarPlus` remains only for `ScalarExpression` instaces:

```
>>> braket = KetSymbol('Psi', hs=0).dag() * KetSymbol('Phi', hs=0)
>>> print(srepr(braket + 1, indented=True))
ScalarPlus(
    One,
    BraKet(
        KetSymbol(
            'Psi',
            hs=LocalSpace(
                '0')),
        KetSymbol(
            'Phi',
            hs=LocalSpace(
                '0'))))
```

simplifications = [<function assoc>, <function convert_to_scalars>, <function orderby>

conjugate()
Complex conjugate of of the sum

class qnet.algebra.core.scalar_algebra.**ScalarTimes**(**operands*, ***kwargs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumTimes*, *qnet.algebra.core.scalar_algebra.Scalar*

Product of scalars

Generally, *ScalarValue* instances are combined directly:

```
>>> alpha = ScalarValue.create(sympy.symbols('alpha'))
>>> print(srepr(alpha * 2))
ScalarValue(Mul(Integer(2), Symbol('alpha')))
```

An unevaluated *ScalarTimes* remains only for *ScalarExpression* instaces:

```
>>> braket = KetSymbol('Psi', hs=0).dag() * KetSymbol('Phi', hs=0)
>>> print(srepr(braket * 2, indented=True))
ScalarTimes(
    ScalarValue(
        2),
    BraKet(
        KetSymbol(
            'Psi',
            hs=LocalSpace(
                '0')),
        KetSymbol(
            'Phi',
            hs=LocalSpace(
                '0'))))
```

simplifications = [<function assoc>, <function orderby>, <function filter_neutral>, <f

classmethod create(**operands*, ***kwargs*)
Instantiate the product while applying simplification rules

conjugate()
Complex conjugate of of the product

class qnet.algebra.core.scalar_algebra.**ScalarIndexedSum**(*term*, **ranges*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumIndexedSum*, *qnet.algebra.core.scalar_algebra.Scalar*

Indexed sum over scalars

simplifications = [<function assoc_indexed>, <function indexed_sum_over_kronecker>, <f

classmethod create (*term*, **ranges*)
Instantiate the indexed sum while applying simplification rules

conjugate ()
Complex conjugate of of the indexed sum

real
Real part

imag
Imaginary part

class qnet.algebra.core.scalar_algebra.**ScalarPower** (*b*, *e*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumOperation*, *qnet.algebra.core.scalar_algebra.Scalar*

A scalar raised to a power

Generally, *ScalarValue* instances are exponentiated directly:

```
>>> alpha = ScalarValue.create(sympy.symbols('alpha'))
>>> print(srepr(alpha**2))
ScalarValue(Pow(Symbol('alpha'), Integer(2)))
```

An unevaluated *ScalarPower* remains only for *ScalarExpression* instaces, see e.g. *sqrt()*.

simplifications = [<function convert_to_scalars>, <function match_replace>]

base
The base of the exponential

exp
The exponent

class qnet.algebra.core.scalar_algebra.**ScalarDerivative** (*op*, ***, *derivs*, *vals=None*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumDerivative*, *qnet.algebra.core.scalar_algebra.Scalar*

Symbolic partial derivative of a scalar

See *QuantumDerivative*.

qnet.algebra.core.scalar_algebra.**KroneckerDelta** (*i*, *j*, *simplify=True*)
Kronecker delta symbol

Return *One* (*i* equals *j*)), *Zero* (*i* and *j* are non-symbolic an unequal), or a *ScalarValue* wrapping SymPy's `KroneckerDelta`.

```
>>> i, j = IdxSym('i'), IdxSym('j')
>>> KroneckerDelta(i, i)
One
>>> KroneckerDelta(1, 2)
Zero
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
```

By default, the Kronecker delta is returned in a simplified form, e.g:

```
>>> KroneckerDelta((i+1)/2, (j+1)/2)
KroneckerDelta(i, j)
```

This may be suppressed by setting *simplify* to False:

```
>>> KroneckerDelta((i+1)/2, (j+1)/2, simplify=False)
KroneckerDelta(i/2 + 1/2, j/2 + 1/2)
```

Raises

- `TypeError` – if *i* or *j* is not an integer or sympy expression. There
- is no automatic sympification of *i* and *j*.

`qnet.algebra.core.scalar_algebra.`**`sqrt`**(*scalar*)

Square root of a `Scalar` or scalar value

This always returns a `Scalar`, and uses a symbolic square root if possible (i.e., for non-floats):

```
>>> sqrt(2)
sqrt(2)

>>> sqrt(2.0)
1.414213...
```

For a `ScalarExpression` argument, it returns a `ScalarPower` instance:

```
>>> braket = KetSymbol('Psi', hs=0).dag() * KetSymbol('Phi', hs=0)
>>> nrm = sqrt(braket * braket.dag())
>>> print(srepr(nrm, indented=True))
ScalarPower(
    ScalarTimes(
        BraKet(
            KetSymbol(
                'Phi',
                hs=LocalSpace(
                    '0')),
            KetSymbol(
                'Psi',
                hs=LocalSpace(
                    '0'))),
        BraKet(
            KetSymbol(
                'Psi',
                hs=LocalSpace(
                    '0')),
            KetSymbol(
                'Phi',
                hs=LocalSpace(
                    '0')))),
    ScalarValue(
        Rational(1, 2)))
```

`qnet.algebra.core.scalar_algebra.`**`is_scalar`**(*scalar*)

Check if *scalar* is a `Scalar` or a scalar value

Specifically, whether *scalar* is an instance of `Scalar` or an instance of a numeric or symbolic type that could be wrapped in `ScalarValue`.

For internal use only.

qnet.algebra.core.state_algebra module

This module implements the algebra of states in a Hilbert space

For more details see *State (Ket-) Algebra*.

Summary

Classes:

`BasisKet`	Local basis state, identified by index or label
`Bra`	The associated dual/adjoint state for any ket
`BraKet`	The symbolic inner product between two states
`CoherentStateKet`	Local coherent state, labeled by a complex amplitude
`KetBra`	Outer product of two states
`KetIndexedSum`	Indexed sum over Kets
`KetPlus`	Sum of states
`KetSymbol`	Symbolic state
`LocalKet`	A state on a `LocalSpace`
`OperatorTimesKet`	Product of an operator and a state.
`ScalarTimesKet`	Product of a `Scalar` coefficient and a ket
`State`	Base class for states in a Hilbert space
`StateDerivative`	Symbolic partial derivative of a state
`TensorKet`	A tensor product of kets

Data:

`TrivialKet`	TrivialKet constant (singleton) object.
`ZeroKet`	ZeroKet constant (singleton) object for the null-state.

`__all__`: `BasisKet`, `Bra`, `BraKet`, `CoherentStateKet`, `KetBra`, `KetIndexedSum`, `KetPlus`, `KetSymbol`, `LocalKet`, `OperatorTimesKet`, `ScalarTimesKet`, `State`, `StateDerivative`, `TensorKet`, `TrivialKet`, `ZeroKet`

Reference

class `qnet.algebra.core.state_algebra.`**`State`**(**args*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumExpression`

Base class for states in a Hilbert space

`isket`
Whether the state represents a ket

`isbra`
Wether the state represents a bra (adjoint ket)

`bra`
The bra associated with a ket

ket
The ket associated with a bra

class qnet.algebra.core.state_algebra.**KetSymbol**(*label*, **sym_args*, *hs*)
Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumSymbol*, *qnet.algebra.core.state_algebra.State*

Symbolic state

See *QuantumSymbol*.

class qnet.algebra.core.state_algebra.**LocalKet**(**args*, *hs*)
Bases: *qnet.algebra.core.state_algebra.State*

A state on a `LocalSpace`

This does not include operations, even if these operations only involve states acting on the same local space

space
The *HilbertSpace* on which the operator acts non-trivially

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

qnet.algebra.core.state_algebra.**ZeroKet = ZeroKet**
ZeroKet constant (singleton) object for the null-state.

qnet.algebra.core.state_algebra.**TrivialKet = TrivialKet**
TrivialKet constant (singleton) object. This is the neutral element under the state tensor-product.

class qnet.algebra.core.state_algebra.**BasisKet**(*label_or_index*, **, *hs*)
Bases: *qnet.algebra.core.state_algebra.LocalKet*, *qnet.algebra.core.state_algebra.KetSymbol*

Local basis state, identified by index or label

Basis kets are orthornormal, and the *next()* and *prev()* methods can be used to move between basis states.

Parameters

- **label_or_index** – If *str*, the label of the basis state (must be an element of *hs.basis_labels*). If *int*, the (zero-based) index of the basis state. This works if *hs* has an unknown dimension. For a symbolic index, *label_or_index* can be an instance of an appropriate subclass of `SymbolicLabelBase`
- **hs** (`LocalSpace`) – The Hilbert space in which the basis is defined

Raises

- `ValueError` – if *label_or_index* is not in the Hilbert space
- `TypeError` – if *label_or_index* is not of an appropriate type
- *BasisNotSetError* – if *label_or_index* is a *str* but no basis is defined for *hs*

Note: Basis states that are instantiated via a label or via an index are equivalent:

```
>>> hs = LocalSpace('tls', basis=('g', 'e'))
>>> BasisKet('g', hs=hs) == BasisKet(0, hs=hs)
True
>>> print(ascii(BasisKet(0, hs=hs)))
|g>^(tls)
```

When instantiating the *BasisKet* via `create()`, an integer label outside the range of the underlying Hilbert space results in a *ZeroKet*:

```
>>> BasisKet.create(-1, hs=0)
ZeroKet
>>> BasisKet.create(2, hs=LocalSpace('tls', dimension=2))
ZeroKet
```

---

simplifications = [<function basis_ket_zero_outside_hs>]

args
Tuple containing *label_or_index* as its only element.

index
The index of the state in the Hilbert space basis

```
>>> hs = LocalSpace('tls', basis=('g', 'e'))
>>> BasisKet('g', hs=hs).index
0
>>> BasisKet('e', hs=hs).index
1
>>> BasisKet(1, hs=hs).index
1
```

For a *BasisKet* with an indexed label, this may return a sympy expression:

```
>>> hs = SpinSpace('s', spin='3/2')
>>> i = symbols('i', cls=IdxSym)
>>> lbl = SpinIndex(i/2, hs)
>>> ket = BasisKet(lbl, hs=hs)
>>> ket.index
```

i/2 + 3/2

next (*n=1*)
Move up by *n* steps in the Hilbert space:

```
>>> hs =  LocalSpace('tls', basis=('g', 'e'))
>>> ascii(BasisKet('g', hs=hs).next())
'|e>^(tls)'
>>> ascii(BasisKet(0, hs=hs).next())
'|e>^(tls)'
```

We can also go multiple steps:

```
>>> hs =  LocalSpace('ten', dimension=10)
>>> ascii(BasisKet(0, hs=hs).next(2))
'|2>^(ten)'
```

An increment that leads out of the Hilbert space returns zero:

```
>>> BasisKet(0, hs=hs).next(10)
ZeroKet
```

prev (*n=1*)
Move down by *n* steps in the Hilbert space, cf. *next()*.

```
>>> hs =  LocalSpace('3l', basis=('g', 'e', 'r'))
>>> ascii(BasisKet('r', hs=hs).prev(2))
'|g>^(3l)'
>>> BasisKet('r', hs=hs).prev(3)
ZeroKet
```

class qnet.algebra.core.state_algebra.**CoherentStateKet**(*ampl*, ***, *hs*)
Bases: *qnet.algebra.core.state_algebra.LocalKet*

Local coherent state, labeled by a complex amplitude

Parameters

- **hs** (LocalSpace) – The local Hilbert space degree of freedom.
- **ampl** (Scalar) – The coherent displacement amplitude.

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

ampl

to_fock_representation(*index_symbol='n'*, *max_terms=None*)
Return the coherent state written out as an indexed sum over Fock basis states

class qnet.algebra.core.state_algebra.**KetPlus**(**operands*)
Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_quantum_algebra.QuantumPlus*

Sum of states

simplifications = [<function accept_bras>, <function assoc>, <function orderby>, <func

order_key
alias of *qnet.utils.ordering.FullCommutativeHSOrder*

class qnet.algebra.core.state_algebra.**TensorKet**(**operands*)
Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_quantum_algebra.QuantumTimes*

A tensor product of kets

Each ket must belong to different degree of freedom (*LocalSpace*).

simplifications = [<function accept_bras>, <function assoc>, <function orderby>, <func

order_key
alias of *qnet.utils.ordering.FullCommutativeHSOrder*

classmethod create(**ops*)
Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are *match_replace()* and *match_replace_binary()* which apply rule-based simplifications.

The *temporary_rules()* context manager may be used to allow temporary modification of the automatic simplifications that *create()* uses, in particular the rules for *match_replace()* and *match_replace_binary()*. Inside the managed context, the *simplifications* class attribute may be modified and rules can be managed with add_rule() and del_rules().

class qnet.algebra.core.state_algebra.**ScalarTimesKet**(*coeff*, *term*)

Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_quantum_algebra.ScalarTimesQuantumExpression*

Product of a *Scalar* coefficient and a ket

Parameters

- **coeff** (Scalar) – coefficient
- **term** (State) – the ket that is multiplied

simplifications = [<function match_replace>]

classmethod create(*coeff*, *term*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are *match_replace()* and *match_replace_binary()* which apply rule-based simplifications.

The *temporary_rules()* context manager may be used to allow temporary modification of the automatic simplifications that *create()* uses, in particular the rules for *match_replace()* and *match_replace_binary()*. Inside the managed context, the *simplifications* class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

class qnet.algebra.core.state_algebra.**OperatorTimesKet**(*operator*, *ket*)

Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_algebra.Operation*

Product of an operator and a state.

simplifications = [<function match_replace>]

space

The *HilbertSpace* on which the operator acts non-trivially

operator

ket

The ket associated with a bra

class qnet.algebra.core.state_algebra.**StateDerivative**(*op*, *, *derivs*, *vals=None*)

Bases: *qnet.algebra.core.abstract_quantum_algebra.QuantumDerivative*, *qnet.algebra.core.state_algebra.State*

Symbolic partial derivative of a state

See *QuantumDerivative*.

class qnet.algebra.core.state_algebra.**Bra**(*ket*)

Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_quantum_algebra.QuantumAdjoint*

The associated dual/adjoint state for any ket

ket

The original *State*

bra

The bra associated with a ket

operand
The original *State*

isket
False, by defintion

isbra
True, by definition

label

class qnet.algebra.core.state_algebra.**BraKet**(*bra*, *ket*)
Bases: *qnet.algebra.core.scalar_algebra.ScalarExpression*, *qnet.algebra.core.abstract_algebra.Operation*

The symbolic inner product between two states

This mathermatically corresponds to:

$$\langle b|k\rangle$$

which we define to be linear in the state k and anti-linear in b.

Parameters

- **bra** (State) – The anti-linear state argument. Note that this is *not* a *Bra* instance.
- **ket** (State) – The linear state argument.

simplifications = [<function match_replace>]

ket
The ket of the braket

bra
The bra of the braket (*Bra* instance)

class qnet.algebra.core.state_algebra.**KetBra**(*ket*, *bra*)
Bases: *qnet.algebra.core.operator_algebra.Operator*, *qnet.algebra.core.abstract_algebra.Operation*

Outer product of two states

Parameters

- **ket** (State) – The left factor in the product
- **bra** (State) – The right factor in the product. Note that this is *not* a *Bra* instance.

simplifications = [<function match_replace>]

ket
The left factor in the product

bra
The co-state right factor in the product

This is a *Bra* instance (unlike the *bra* given to the constructor

space
The Hilbert space of the states being multiplied

class qnet.algebra.core.state_algebra.**KetIndexedSum**(*term*, **ranges*)
Bases: *qnet.algebra.core.state_algebra.State*, *qnet.algebra.core.abstract_quantum_algebra.QuantumIndexedSum*

Indexed sum over Kets

simplifications = [<function assoc_indexed>, <function indexed_sum_over_kronecker>, <f

classmethod create(*term*, **ranges*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

qnet.algebra.core.super_operator_algebra module

The specification of a quantum mechanics symbolic super-operator algebra. See *Super-Operator Algebra* for more details.

Summary

Classes:

`SPost`	Linear post-multiplication operator
`SPre`	Linear pre-multiplication operator
`ScalarTimesSuperOperator`	Product of a `Scalar` coefficient and a `SuperOperator`
`SuperAdjoint`	Adjoint of a super-operator
`SuperCommutativeHSOrder`	Ordering class that acts like DisjunctCommutativeHSOrder, but also commutes any *SPost* and *SPre*
`SuperOperator`	Base class for super-operators
`SuperOperatorDerivative`	Symbolic partial derivative of a super-operator
`SuperOperatorPlus`	A sum of super-operators
`SuperOperatorSymbol`	Symbolic super-operator
`SuperOperatorTimes`	Product of super-operators
`SuperOperatorTimesOperator`	Application of a super-operator to an operator

Functions:

`anti_commutator`	If `B != None`, return the anti-commutator $\{A, B\}$, otherwise return the super-operator $\{A, \cdot\}$.
`commutator`	Commutator of *A* and *B*
`lindblad`	Return the super-operator Lindblad term of the Lindblad operator *C*
`liouvillian`	Return the Liouvillian super-operator associated with *H* and *Ls*
`liouvillian_normal_form`	Return a Hamilton operator `H` and a minimal list of collapse operators `Ls` that generate the liouvillian `L`.

Data:

`IdentitySuperOperator`	Neutral element for product of super-operators
`ZeroSuperOperator`	Neutral element for sum of super-operators

`__all__`: `IdentitySuperOperator`, `SPost`, `SPre`, `ScalarTimesSuperOperator`, `SuperAdjoint`, `SuperOperator`, `SuperOperatorDerivative`, `SuperOperatorPlus`, `SuperOperatorSymbol`, `SuperOperatorTimes`, `SuperOperatorTimesOperator`, `ZeroSuperOperator`, `anti_commutator`, `commutator`, `lindblad`, `liouvillian`, `liouvillian_normal_form`

Reference

class `qnet.algebra.core.super_operator_algebra.`**SuperOperator**(**args*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumExpression`

Base class for super-operators

class `qnet.algebra.core.super_operator_algebra.`**SuperOperatorSymbol**(*label*, **sym_args*, *hs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumSymbol`, `qnet.algebra.core.super_operator_algebra.SuperOperator`

Symbolic super-operator

See `QuantumSymbol`.

`qnet.algebra.core.super_operator_algebra.`**IdentitySuperOperator = IdentitySuperOperator**

Neutral element for product of super-operators

`qnet.algebra.core.super_operator_algebra.`**ZeroSuperOperator = ZeroSuperOperator**

Neutral element for sum of super-operators

class `qnet.algebra.core.super_operator_algebra.`**SuperOperatorPlus**(**operands*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumPlus`, `qnet.algebra.core.super_operator_algebra.SuperOperator`

A sum of super-operators

simplifications = [<function assoc>, <function orderby>, <function collect_summands>,

class `qnet.algebra.core.super_operator_algebra.`**SuperCommutativeHSOrder**(*op*, *space_order=None*, *op_order=None*)

Bases: `qnet.utils.ordering.DisjunctCommutativeHSOrder`

Ordering class that acts like DisjunctCommutativeHSOrder, but also commutes any *SPost* and *SPre*

class `qnet.algebra.core.super_operator_algebra.`**SuperOperatorTimes**(**operands*, ***kwargs*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumTimes`, `qnet.algebra.core.super_operator_algebra.SuperOperator`

Product of super-operators

simplifications = [<function assoc>, <function orderby>, <function filter_neutral>, <f

order_key

alias of `SuperCommutativeHSOrder`

classmethod create(**ops*)

Instantiate while applying automatic simplifications

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

Two simplifications of particular importance are `match_replace()` and `match_replace_binary()` which apply rule-based simplifications.

The `temporary_rules()` context manager may be used to allow temporary modification of the automatic simplifications that `create()` uses, in particular the rules for `match_replace()` and `match_replace_binary()`. Inside the managed context, the `simplifications` class attribute may be modified and rules can be managed with `add_rule()` and `del_rules()`.

class qnet.algebra.core.super_operator_algebra.**ScalarTimesSuperOperator**(*coeff*, *term*)

Bases: `qnet.algebra.core.super_operator_algebra.SuperOperator`, `qnet.algebra.core.abstract_quantum_algebra.ScalarTimesQuantumExpression`

Product of a `Scalar` coefficient and a `SuperOperator`

simplifications = [<function match_replace>]

class qnet.algebra.core.super_operator_algebra.**SuperAdjoint**(*operand*)

Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumAdjoint`, `qnet.algebra.core.super_operator_algebra.SuperOperator`

Adjoint of a super-operator

The mathematical notation for this is typically

$$\text{SuperAdjoint}(\mathcal{L}) =: \mathcal{L}^*$$

and for any super operator $\mathcal{L}$, its super-adjoint $\mathcal{L}^*$ satisfies for any pair of operators M, N:

$$\text{Tr}[M(\mathcal{L}N)] = Tr[(\mathcal{L}^*M)N]$$

simplifications = [<function delegate_to_method.<locals>._delegate_to_method>]

class qnet.algebra.core.super_operator_algebra.**SPre**(**args*, ***kwargs*)

Bases: `qnet.algebra.core.super_operator_algebra.SuperOperator`, `qnet.algebra.core.abstract_algebra.Operation`

Linear pre-multiplication operator

Acting `SPre(A)` on an operator `B` just yields the product `A * B`

simplifications = [<function match_replace>]

space

The `HilbertSpace` on which the operator acts non-trivially

class qnet.algebra.core.super_operator_algebra.**SPost**(**args*, ***kwargs*)

Bases: `qnet.algebra.core.super_operator_algebra.SuperOperator`, `qnet.algebra.core.abstract_algebra.Operation`

Linear post-multiplication operator

Acting `SPost(A)` on an operator `B` just yields the reversed product `B * A`.

simplifications = [<function match_replace>]

space
The `HilbertSpace` on which the operator acts non-trivially

class qnet.algebra.core.super_operator_algebra.**SuperOperatorTimesOperator**(*sop*, *op*)
Bases: `qnet.algebra.core.operator_algebra.Operator`, `qnet.algebra.core.abstract_algebra.Operation`

Application of a super-operator to an operator

The result of this operation is(result is an `Operator`

simplifications = [<function match_replace>]

space
The `HilbertSpace` on which the operator acts non-trivially

sop

op

class qnet.algebra.core.super_operator_algebra.**SuperOperatorDerivative**(*op*, **, *derivs*, *vals=None*)
Bases: `qnet.algebra.core.abstract_quantum_algebra.QuantumDerivative`, `qnet.algebra.core.super_operator_algebra.SuperOperator`

Symbolic partial derivative of a super-operator

See `QuantumDerivative`.

qnet.algebra.core.super_operator_algebra.**commutator**(*A*, *B=None*)
Commutator of A and B

If `B != None`, return the commutator $[A, B]$, otherwise return the super-operator $[A, \cdot]$. The super-operator $[A, \cdot]$ maps any other operator `B` to the commutator $[A, B] = AB - BA$.

Parameters

- **A** – The first operator to form the commutator of.
- **B** – The second operator to form the commutator of, or None.

Returns The linear superoperator $[A, \cdot]$

Return type *SuperOperator*

qnet.algebra.core.super_operator_algebra.**anti_commutator**(*A*, *B=None*)
If `B != None`, return the anti-commutator $\{A, B\}$, otherwise return the super-operator $\{A, \cdot\}$. The super-operator $\{A, \cdot\}$ maps any other operator `B` to the anti-commutator $\{A, B\} = AB + BA$.

Parameters

- **A** – The first operator to form all anti-commutators of.
- **B** – The second operator to form the anti-commutator of, or None.

Returns The linear superoperator $[A, \cdot]$

Return type *SuperOperator*

qnet.algebra.core.super_operator_algebra.**lindblad**(*C*)
Return the super-operator Lindblad term of the Lindblad operator C

Return `SPre(C) * SPost(C.adjoint()) - (1/2) * santi_commutator(C.adjoint()*C)`. These are the super-operators $\mathcal{D}[C]$ that form the collapse terms of a Master-Equation. Applied to an operator X they yield

$$\mathcal{D}[C]X = CXC^\dagger - \frac{1}{2}(C^\dagger CX + XC^\dagger C)$$

Parameters **C** (`Operator`) – The associated collapse operator

Returns The Lindblad collapse generator.

Return type *SuperOperator*

`qnet.algebra.core.super_operator_algebra.`**`liouvillian`**`(H, Ls=None)`

Return the Liouvillian super-operator associated with H and Ls

The Liouvillian $\mathcal{L}$ generates the Markovian-dynamics of a system via the Master equation:

$$\dot{\rho} = \mathcal{L}\rho = -i[H,\rho] + \sum_{j=1}^{n} \mathcal{D}[L_j]\rho$$

Parameters

- **H** (`Operator`) – The associated Hamilton operator
- **Ls** (*sequence or* `Matrix`) – A sequence of Lindblad operators.

Returns The Liouvillian super-operator.

Return type *SuperOperator*

`qnet.algebra.core.super_operator_algebra.`**`liouvillian_normal_form`**`(L, symbolic=False)`

Return a Hamilton operator `H` and a minimal list of collapse operators `Ls` that generate the liouvillian `L`.

A Liouvillian defined by a hermitian Hamilton operator H and a vector of collapse operators $\mathbf{L} = (L_1, L_2, \dots L_n)^T$ is invariant under the following two operations:

$$(H, \mathbf{L}) \mapsto \left(H + \frac{1}{2i}\left(\mathbf{w}^\dagger \mathbf{L} - \mathbf{L}^\dagger \mathbf{w}\right), \mathbf{L} + \mathbf{w}\right)$$

$$(H, \mathbf{L}) \mapsto (H, \mathbf{UL})$$

where $\mathbf{w}$ is just a vector of complex numbers and $\mathbf{U}$ is a complex unitary matrix. It turns out that for quantum optical circuit models the set of collapse operators is often linearly dependent. This routine tries to find a representation of the Liouvillian in terms of a Hamilton operator `H` with as few non-zero collapse operators `Ls` as possible. Consider the following example, which results from a two-port linear cavity with a coherent input into the first port:

```
>>> kappa_1, kappa_2 = sympy.symbols('kappa_1, kappa_2', positive = True)
>>> Delta = sympy.symbols('Delta', real = True)
>>> alpha = sympy.symbols('alpha')
>>> H = (Delta * Create(hs=1) * Destroy(hs=1) +
...      (sqrt(kappa_1) / (2 * I)) *
...      (alpha * Create(hs=1) - alpha.conjugate() * Destroy(hs=1)))
>>> Ls = [sqrt(kappa_1) * Destroy(hs=1) + alpha,
...       sqrt(kappa_2) * Destroy(hs=1)]
>>> LL = liouvillian(H, Ls)
>>> Hnf, Lsnf = liouvillian_normal_form(LL)
>>> print(ascii(Hnf))
-I*alpha*sqrt(kappa_1) * a^(1)H + I*sqrt(kappa_1)*conjugate(alpha) * a^(1) +
↪Delta * a^(1)H * a^(1)
```

(continues on next page)

(continued from previous page)

```
>>> len(Lsnf)
1
>>> print(ascii(Lsnf[0]))
sqrt(kappa_1 + kappa_2) * a^(1)
```

In terms of the ensemble dynamics this final system is equivalent. Note that this function will only work for proper Liouvillians.

Parameters `L` (`SuperOperator`) – The Liouvillian

Returns `(H, Ls)`

Return type tuple

Raises *BadLiouvillianError*

Summary

`__all__` Exceptions:

AlgebraError	Base class for all algebraic errors
AlgebraException	Base class for all algebraic exceptions
BadLiouvillianError	Raised when a Liouvillian is not of standard Lindblad form.
BasisNotSetError	Raised if the basis or a Hilbert space dimension is unavailable
CannotConvertToSLH	Raised when a circuit algebra object cannot be converted to SLH
CannotEliminateAutomatically	Raised when attempted automatic adiabatic elimination fails.
CannotSimplify	Raised when a rule cannot further simplify an expression
CannotSymbolicallyDiagonalize	Matrix cannot be diagonalized analytically.
CannotVisualize	Raised when a circuit cannot be visually represented.
IncompatibleBlockStructures	Raised for invalid block-decomposition
InfiniteSumError	Raised when expanding a sum into an infinite number of terms
NoConjugateMatrix	Raised when entries of *Matrix* have no defined conjugate
NonSquareMatrix	Raised when a *Matrix* fails to be square
OverlappingSpaces	Raised when objects fail to be in separate Hilbert spaces.
SpaceTooLargeError	Raised when objects fail to be have overlapping Hilbert spaces.
UnequalSpaces	Raised when objects fail to be in the same Hilbert space.
WrongCDimError	Raised for mismatched channel number in circuit series

`__all__` Classes:

Adjoint	Symbolic Adjoint of an operator
BasisKet	Local basis state, identified by index or label
Bra	The associated dual/adjoint state for any ket
BraKet	The symbolic inner product between two states
CPermutation	Channel permuting circuit
Circuit	Base class for the circuit algebra elements
CircuitSymbol	Symbolic circuit element
CoherentStateKet	Local coherent state, labeled by a complex amplitude
Commutator	Commutator of two operators
Component	Base class for circuit components
Concatenation	Concatenation of circuit elements

Continued on next page

Table 26 – continued from previous page

Expression	Base class for all QNET Expressions
Feedback	Feedback on a single channel of a circuit
HilbertSpace	Base class for Hilbert spaces
IndexedSum	Base class for indexed sums
KetBra	Outer product of two states
KetIndexedSum	Indexed sum over Kets
KetPlus	Sum of states
KetSymbol	Symbolic state
LocalKet	A state on a LocalSpace
LocalOperator	Base class for "known" operators on a LocalSpace
LocalSigma	Level flip operator between two levels of a LocalSpace
LocalSpace	Hilbert space for a single degree of freedom.
Matrix	Matrix of Expressions
NullSpaceProjector	Projection operator onto the nullspace of its operand
Operation	Base class for "operations"
Operator	Base class for all quantum operators.
OperatorDerivative	Symbolic partial derivative of an operator
OperatorIndexedSum	Indexed sum over operators
OperatorPlus	Sum of Operators
OperatorPlusMinusCC	An operator plus or minus its complex conjugate
OperatorSymbol	Symbolic operator
OperatorTimes	Product of operators
OperatorTimesKet	Product of an operator and a state.
OperatorTrace	(Partial) trace of an operator
ProductSpace	Tensor product of local Hilbert spaces
PseudoInverse	Unevaluated pseudo-inverse X^+ of an operator X
QuantumAdjoint	Base class for adjoints of quantum expressions
QuantumDerivative	Symbolic partial derivative
QuantumExpression	Base class for expressions associated with a Hilbert space
QuantumIndexedSum	Base class for indexed sums
QuantumOperation	Base class for operations on quantum expression
QuantumPlus	General implementation of addition of quantum expressions
QuantumSymbol	Symbolic element of an algebra
QuantumTimes	General implementation of product of quantum expressions
SLH	Element of the SLH algebra
SPost	Linear post-multiplication operator
SPre	Linear pre-multiplication operator
Scalar	Base class for Scalars
ScalarDerivative	Symbolic partial derivative of a scalar
ScalarExpression	Base class for scalars with non-scalar arguments
ScalarIndexedSum	Indexed sum over scalars
ScalarPlus	Sum of scalars
ScalarPower	A scalar raised to a power
ScalarTimes	Product of scalars
ScalarTimesKet	Product of a Scalar coefficient and a ket
ScalarTimesOperator	Product of a Scalar coefficient and an Operator
ScalarTimesQuantumExpression	Product of a Scalar and a QuantumExpression
ScalarTimesSuperOperator	Product of a Scalar coefficient and a SuperOperator
ScalarValue	Wrapper around a numeric or symbolic value
SeriesInverse	Symbolic series product inversion operation

Continued on next page

Table 26 – continued from previous page

SeriesProduct	The series product circuit operation.
SingleQuantumOperation	Base class for operations on a single quantum expression
State	Base class for states in a Hilbert space
StateDerivative	Symbolic partial derivative of a state
SuperAdjoint	Adjoint of a super-operator
SuperOperator	Base class for super-operators
SuperOperatorDerivative	Symbolic partial derivative of a super-operator
SuperOperatorPlus	A sum of super-operators
SuperOperatorSymbol	Symbolic super-operator
SuperOperatorTimes	Product of super-operators
SuperOperatorTimesOperator	Application of a super-operator to an operator
TensorKet	A tensor product of kets

`__all__` Functions:

FB	Wrapper for Feedback, defaulting to last channel
KroneckerDelta	Kronecker delta symbol
LocalProjector	A projector onto a specific level of a LocalSpace
Sum	Instantiator for an arbitrary indexed sum.
adjoint	Return the adjoint of an obj.
anti_commutator	If `B != None`, return the anti-commutator $\{A, B\}$, otherwise return the super-operator $\{$
block_matrix	Generate the operator matrix with quadrants
circuit_identity	Return the circuit identity for n channels
commutator	Commutator of A and B
decompose_space	Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated
diagm	Generalizes the diagonal matrix creation capabilities of *numpy.diag* to `Matrix` objects.
eval_adiabatic_limit	Compute the limiting SLH model for the adiabatic approximation
extract_channel	Create a `CPermutation` that extracts channel k
factor_coeff	Factor out coefficients of all factors.
factor_for_trace	Given a LocalSpace *ls* to take the partial trace over and an operator *op*, factor the trace
getABCD	Calculate the ABCD-linearization of an SLH model
get_coeffs	Create a dictionary with all Operator terms of the expression (understood as a sum) as key
hstackm	Generalizes *numpy.hstack* to `Matrix` objects.
identity_matrix	Generate the N-dimensional identity matrix.
lindblad	Return the super-operator Lindblad term of the Lindblad operator C
liouvillian	Return the Liouvillian super-operator associated with H and Ls
liouvillian_normal_form	Return a Hamilton operator `H` and a minimal list of collapse operators `Ls` that generate the
map_channels	Create a `CPermuation` based on a dict of channel mappings
move_drive_to_H	Move coherent drives from the Lindblad operators to the Hamiltonian.
pad_with_identity	Pad a circuit by adding a n-channel identity circuit at index k
prepare_adiabatic_limit	Prepare the adiabatic elimination on an SLH object
rewrite_with_operator_pm_cc	Try to rewrite expr using `OperatorPlusMinusCC`
sqrt	Square root of a `Scalar` or scalar value
substitute	Substitute symbols or (sub-)expressions with the given replacements and re-evalute the res
try_adiabatic_elimination	Attempt to automatically do adiabatic elimination on an SLH object
vstackm	Generalizes *numpy.vstack* to `Matrix` objects.
zerosm	Generalizes `numpy.zeros` to `Matrix` objects.

`__all__` Data:

CIdentity	Single pass-through channel; neutral element of `SeriesProduct`
CircuitZero	Zero circuit, the neutral element of `Concatenation`
FullSpace	The 'full space', i.e.
II	`IdentityOperator` constant (singleton) object.
IdentityOperator	`IdentityOperator` constant (singleton) object.
IdentitySuperOperator	Neutral element for product of super-operators
One	The neutral element with respect to scalar multiplication
TrivialKet	TrivialKet constant (singleton) object.
TrivialSpace	The 'nullspace', i.e.
Zero	The neutral element with respect to scalar addition
ZeroKet	ZeroKet constant (singleton) object for the null-state.
ZeroOperator	`ZeroOperator` constant (singleton) object.
ZeroSuperOperator	Neutral element for sum of super-operators
tr	Instantiate while applying automatic simplifications

qnet.algebra.library package

Collection of algebraic objects extending *core*

Submodules:

qnet.algebra.library.circuit_components module

Collection of essential circuit components

Summary

Classes:

Beamsplitter	Infinite bandwidth beamsplitter component.
CoherentDriveCC	Coherent displacement of the input field
PhaseCC	Coherent phase shift cicuit component

`__all__`: *Beamsplitter*, *CoherentDriveCC*, *PhaseCC*

Reference

class qnet.algebra.library.circuit_components.**CoherentDriveCC**(*, *label=None*, ***kwargs*)

Bases: *qnet.algebra.core.circuit_algebra.Component*

Coherent displacement of the input field

Typically, the input field is the, displaced by a complex amplitude α. This component serves as the model of an ideal laser source without internal non-classical internal dynamics.

The coherent drive is represented as an inhomogeneous Lindblad operator $L = \alpha$, with a trivial Hamiltonian and scattering matrix. For a complete circuit with coherent drives, the inhomogeneous Lindblad operators can be transformed to driving terms in the total network Hamiltonian through *move_drive_to_H()*.

Parameters

- **label** – label for the component.
- **displacement** – the coherent displacement amplitude. Defaults to a complex symbol 'alpha'

CDIM = 1
circuit dimension

PORTSIN = ('in',)

PORTSOUT = ('out',)

ARGNAMES = ('displacement',)

DEFAULTS = {'displacement': alpha}

IDENTIFIER = 'W'

displacement
The *displacement* argument.

class qnet.algebra.library.circuit_components.**PhaseCC**(**, label=None, **kwargs*)
Bases: *qnet.algebra.core.circuit_algebra.Component*

Coherent phase shift cicuit component

The field passing through obtains a phase factor $e^{i\phi}$ for a real-valued phase ϕ. The component has no dynamics, i.e. a trivial Hamiltonian and Lindblad operators

Parameters

- **label** – label for the component.
- **phase** – the phase. Defaults to a real symbol 'phi'

CDIM = 1

PORTSIN = ('in',)

PORTSOUT = ('out',)

ARGNAMES = ('phase',)

DEFAULTS = {'phase': phi}

IDENTIFIER = 'Phase'

phase
The *phase* argument.

class qnet.algebra.library.circuit_components.**Beamsplitter**(**, label=None, **kwargs*)
Bases: *qnet.algebra.core.circuit_algebra.Component*

Infinite bandwidth beamsplitter component.

It is a pure scattering component, i.e. it's internal dynamics are not modeled explicitly (trivial Hamiltonian and Lindblad operators). The single real parameter is the *mixing_angle* for the two signals.

$$S = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

The beamsplitter uses the following labeled input/output channels:

```
      │1:vac
      │
0:in  v  0:tr
────>/────>
      │
      │1:rf
      v
```

That is, output channel 0 is the transmission of input channel 0 ("in"), and output channel 1 is the reflection of input channel 0; vice versa for the secondary input channel 1 ("vac": often connected to a vacuum mode). For $\theta = 0$, the beam splitter results in full transmission, and full reflection for $\theta = \pi/2$.

Parameters

- **label** – label for the beamsplitter.
- **mixing_angle** – the angle that determines the ratio of transmission and reflection defaults to $\pi/4$, corresponding to a 50-50-beamsplitter. It is recommended to use a sympy expression for the mixing angle.

Note: We use a real-valued, but asymmetric scattering matrix. A common alternative convention for the beamsplitter is the symmetric scattering matrix

$$S = \begin{pmatrix} \cos\theta & i\sin\theta \\ i\sin\theta & \cos\theta \end{pmatrix}$$

To achieve the symmetric beamsplitter (or any general beamsplitter), the `Beamsplitter` component can be combined with one or more appropriate `PhaseCC` components.

CDIM = 2
circuit dimension

PORTSIN = ('in', 'vac')

PORTSOUT = ('tr', 'rf')

ARGNAMES = ('mixing_angle',)

DEFAULTS = {'mixing_angle': pi/4}

IDENTIFIER = 'BS'

mixing_angle
The *mixing_angle* argument.

qnet.algebra.library.fock_operators module

Collection of operators that act on a bosonic Fock space

Summary

Classes:

`Create`	Bosonic creation operator
`Destroy`	Bosonic annihilation operator

Continued on next page

Table 29 – continued from previous page

Displace	Unitary coherent displacement operator
Phase	Unitary "phase" operator
Squeeze	Unitary squeezing operator

`__all__`: *Create*, *Destroy*, *Displace*, *Phase*, *Squeeze*

Reference

class qnet.algebra.library.fock_operators.**Destroy**(*, *hs*)

Bases: *qnet.algebra.core.operator_algebra.LocalOperator*

Bosonic annihilation operator

It obeys the bosonic commutation relation:

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

identifier

The identifier (symbol) that is used when printing the annihilation operator. This is identical to the identifier of *Create*. A custom identifier for both *Destroy* and *Create* can be set through the *local_identifiers* parameter of the associated Hilbert space:

```
>>> hs_custom = LocalSpace(0, local_identifiers={'Destroy': 'b'})
>>> Create(hs=hs_custom).identifier
'b'
>>> Destroy(hs=hs_custom).identifier
'b'
```

class qnet.algebra.library.fock_operators.**Create**(*, *hs*)

Bases: *qnet.algebra.core.operator_algebra.LocalOperator*

Bosonic creation operator

This is the adjoint of *Destroy*.

identifier

The identifier (symbols) that is used when printing the creation operator. This is identical to the identifier of *Destroy*

class qnet.algebra.library.fock_operators.**Phase**(**args*, *hs*)

Bases: *qnet.algebra.core.operator_algebra.LocalOperator*

Unitary "phase" operator

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

where $a_{\rm hs}$ is the annihilation operator acting on the *LocalSpace* *hs*.

Parameters

- **phase** (Scalar) – the phase ϕ
- **hs** (*HilbertSpace or int or str*) – The Hilbert space on which the operator acts

Printers should represent this operator with the default identifier:

```
>>> Phase._identifier
'Phase'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

simplifications = [<function implied_local_space.<locals>.kwargs_to_local_space>, <func

phase
The *phase* argument, as a `Scalar` instance.

class qnet.algebra.library.fock_operators.**Displace**(**args*, *hs*)
Bases: `qnet.algebra.core.operator_algebra.LocalOperator`

Unitary coherent displacement operator

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

where $a_{\rm hs}$ is the annihilation operator acting on the `LocalSpace` *hs*.

Parameters

- **displacement** (`Scalar`) – the displacement amplitude α
- **hs** (`HilbertSpace or int or str`) – The Hilbert space on which the operator acts

Printers should represent this operator with the default identifier:

```
>>> Displace._identifier
'D'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

simplifications = [<function implied_local_space.<locals>.kwargs_to_local_space>, <func

displacement
The *displacement* argument, as a `Scalar` instance.

class qnet.algebra.library.fock_operators.**Squeeze**(**args*, *hs*)
Bases: `qnet.algebra.core.operator_algebra.LocalOperator`

Unitary squeezing operator

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

where $a_{\rm hs}$ is the annihilation operator acting on the `LocalSpace` *hs*.

Parameters

- **squeezing_factor** (`Scalar`) – the squeezing factor η
- **hs** (`HilbertSpace or int or str`) – The Hilbert space on which the operator acts

Printers should represent this operator with the default identifier:

```
>>> Squeeze._identifier
'Squeeze'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

simplifications = [<function implied_local_space.<locals>.kwargs_to_local_space>, <func

squeezing_factor
The *squeezing_factor* argument, as a `Scalar` instance.

qnet.algebra.library.pauli_matrices module

Constructors for Pauli-Matrix operators on any two levels of a system

Summary

Functions:

PauliX	Pauli-type X-operator
PauliY	Pauli-type Y-operator
PauliZ	Pauli-type Z-operator

`__all__`: *PauliX*, *PauliY*, *PauliZ*

Reference

`qnet.algebra.library.pauli_matrices.`**`PauliX`**`(`*local_space*, *states=None*`)`

Pauli-type X-operator

$$\hat{\sigma}_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

on an arbitrary two-level system.

Parameters

- **local_space** (*str or int or* `LocalSpace`) – Associated Hilbert space. If `str` or `int`, a `LocalSpace` with a matching label will be created.
- **states** (*None or tuple[int or str]*) – The labels for the basis states for the two levels on which the operator acts. If None, the two lowest levels are used.

Returns Local X-operator as a linear combination of `LocalSigma`

Return type *Operator*

`qnet.algebra.library.pauli_matrices.`**`PauliY`**`(`*local_space*, *states=None*`)`

Pauli-type Y-operator

$$\hat{\sigma}_x = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

on an arbitrary two-level system.

See *PauliX()*

`qnet.algebra.library.pauli_matrices.`**`PauliZ`**`(`*local_space*, *states=None*`)`

Pauli-type Z-operator

$$\hat{\sigma}_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

on an arbitrary two-level system.

See *PauliX()*

qnet.algebra.library.spin_algebra module

Definitions for an algebra on spin (angular momentum) Hilbert spaces, both for integer and half-integer spin

Summary

Classes:

`Jminus`	Lowering operator on a spin space
`Jplus`	Raising operator of a spin space
`Jz`	Spin (angular momentum) operator in z-direction
`SpinOperator`	Base class for operators in a spin space
`SpinSpace`	A Hilbert space for an integer or half-integer spin system

Functions:

`Jmjmcoeff`	Eigenvalue of the J_- (`Jminus`) operator
`Jpjmcoeff`	Eigenvalue of the J_+ (`Jplus`) operator
`Jzjmcoeff`	Eigenvalue of the J_z (`Jz`) operator
`SpinBasisKet`	Constructor for a `BasisKet` for a `SpinSpace`

`__all__`: `Jminus`, `Jplus`, `Jz`, `SpinBasisKet`, `SpinOperator`, `SpinSpace`

Reference

class qnet.algebra.library.spin_algebra.**SpinSpace**(*label*, ***, *spin*, *basis=None*, *local_identifiers=None*, *order_index=None*)

Bases: `qnet.algebra.core.hilbert_space_algebra.LocalSpace`

A Hilbert space for an integer or half-integer spin system

For a given spin N, the resulting Hilbert space has dimension $2N + 1$ with levels labeled from $-N$ to $+N$ (as strings)

For an integer spin:

```
>>> hs = SpinSpace(label=0, spin=1)
>>> hs.dimension
3
>>> hs.basis_labels
('-1', '0', '+1')
```

For a half-integer spin:

```
>>> hs = SpinSpace(label=0, spin=sympy.Rational(3, 2))
>>> hs.spin
3/2
>>> hs.dimension
4
>>> hs.basis_labels
('-3/2', '-1/2', '+1/2', '+3/2')
```

For convenience, you may also give *spin* as a tuple or a string:

```
>>> hs = SpinSpace(label=0, spin=(3, 2))
>>> assert hs == SpinSpace(label=0, spin=sympy.Rational(3, 2))
>>> hs = SpinSpace(label=0, spin='3/2')
>>> assert hs == SpinSpace(label=0, spin=(3, 2))
```

You may use custom labels, e.g.:

```
>>> hs = SpinSpace(label='s', spin='1/2', basis=('-', '+'))
>>> hs.basis_labels
('-', '+')
```

The labels "up" and "down" are recognized and printed as the appropriate arrow symbols:

```
>>> hs = SpinSpace(label='s', spin='1/2', basis=('down', 'up'))
>>> unicode(BasisKet('up', hs=hs))
'|↑'
>>> unicode(BasisKet('down', hs=hs))
'|↓'
```

Raises `ValueError` – if *spin* is not an integer or half-integer greater than zero

next_basis_label_or_index(*label_or_index*, *n=1*)

Given the label or index of a basis state, return the label the next basis state.

More generally, if *n* is given, return the *n*'th next basis state label/index; *n* may also be negative to obtain previous basis state labels. Returns a `str` label if *label_or_index* is a `str` or `int`, or a `SpinIndex` if *label_or_index* is a `SpinIndex`.

Parameters

- **label_or_index** (*int or str or SpinIndex*) – If *int*, the zero-based index of a basis state; if *str*, the label of a basis state
- **n** (*int*) – The increment

Raises

- `IndexError` – If going beyond the last or first basis state
- `ValueError` – If *label* is not a label for any basis state in the Hilbert space
- *BasisNotSetError* – If the Hilbert space has no defined basis
- `TypeError` – if *label_or_index* is neither a `str` nor an `int`, nor a `SpinIndex`

Note: This differs from its super-method only by never returning an integer index (which is not accepted when instantiating a `BasisKet` for a *SpinSpace*)

spin

The spin-number associated with the *SpinSpace*

This can be a SymPy integer or a half-integer.

Return type `Rational`

multiplicity

The multiplicity of the Hilbert space, $2S + 1$.

This is equivalent to the dimension:

```
>>> hs = SpinSpace('s', spin=sympy.Rational(3, 2))
>>> hs.multiplicity == 4 == hs.dimension
True
```

Return type `int`

`qnet.algebra.library.spin_algebra.`**`SpinBasisKet`**`(*numer_denom, hs)`
Constructor for a `BasisKet` for a *`SpinSpace`*

For a half-integer spin system:

```
>>> hs = SpinSpace('s', spin=(3, 2))
>>> assert SpinBasisKet(1, 2, hs=hs) == BasisKet("+1/2", hs=hs)
```

For an integer spin system:

```
>>> hs = SpinSpace('s', spin=1)
>>> assert SpinBasisKet(1, hs=hs) == BasisKet("+1", hs=hs)
```

Note that `BasisKet(1, hs=hs)` with an integer index (which would hypothetically refer to `BasisKet("0", hs=hs)` is not allowed for spin systems:

```
>>> BasisKet(1, hs=hs)
Traceback (most recent call last):
    ...
TypeError: label_or_index must be an instance of one of str, SpinIndex; not int
```

Raises

- `TypeError` – if *hs* is not a *`SpinSpace`* or the wrong number of positional arguments is given
- `ValueError` – if any of the positional arguments are out range for the given *hs*

class `qnet.algebra.library.spin_algebra.`**`SpinOperator`**`(*args, hs)`
Bases: *`qnet.algebra.core.operator_algebra.LocalOperator`*

Base class for operators in a spin space

class `qnet.algebra.library.spin_algebra.`**`Jz`**`(*, hs)`
Bases: *`qnet.algebra.library.spin_algebra.SpinOperator`*

Spin (angular momentum) operator in z-direction

J_z is the z component of a general spin operator acting on a particular *`SpinSpace`* *hs* of freedom with well defined spin quantum number s. It is Hermitian:

```
>>> hs = SpinSpace(1, spin=(1, 2))
>>> print(ascii(Jz(hs=hs).adjoint()))
J_z^(1)
```

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

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

>>> print(ascii((Jz(hs=hs) * Jminus(hs=hs) -
```

(continues on next page)

(continued from previous page)

```
...                 Jminus(hs=hs)*Jz(hs=hs)).expand()))
-J_-^(1)

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

Printers should represent this operator with the default identifier:

```
>>> Jz._identifier
'J_z'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

class qnet.algebra.library.spin_algebra.**Jplus**(*, *hs*)

Bases: *qnet.algebra.library.spin_algebra.SpinOperator*

Raising operator of a spin space

$J_{+} = J_x + iJ_y$ is the raising ladder operator of a general spin operator acting on a particular *SpinSpace* *hs* with well defined spin quantum number s. It's adjoint is the lowering operator:

```
>>> hs = SpinSpace(1, spin=(1, 2))
>>> print(ascii(Jplus(hs=hs).adjoint()))
J_-^(1)
```

Jz, *Jplus* and *Jminus* satisfy that angular momentum commutator algebra, see *Jz*

Printers should represent this operator with the default identifier:

```
>>> Jplus._identifier
'J_+'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

class qnet.algebra.library.spin_algebra.**Jminus**(*, *hs*)

Bases: *qnet.algebra.library.spin_algebra.SpinOperator*

Lowering operator on a spin space

$J_{-} = J_x - iJ_y$ is the lowering ladder operator of a general spin operator acting on a particular *SpinSpace* *hs* with well defined spin quantum number s. It's adjoint is the raising operator:

```
>>> hs = SpinSpace(1, spin=(1, 2))
>>> print(ascii(Jminus(hs=hs).adjoint()))
J_+^(1)
```

Jz, *Jplus* and *Jminus* satisfy that angular momentum commutator algebra, see *Jz*.

Printers should represent this operator with the default identifier:

```
>>> Jminus._identifier
'J_-'
```

A custom identifier may be define using *hs*'s *local_identifiers* argument.

qnet.algebra.library.spin_algebra.**Jpjmcoeff**(*ls*, *m*, *shift=False*)
Eigenvalue of the J_+ (*Jplus*) operator

$$J_+ s, m = \sqrt{s(s+1) - m(m+1)} s, m$$

where the multiplicity s is implied by the size of the Hilbert space *ls*: there are $2s+1$ eigenstates with $m = -s, -s+1, \dots, s$.

Parameters

- **ls** (LocalSpace) – The Hilbert space in which the J_+ operator acts.
- **m** (*str or int*) – If str, the label of the basis state of *hs* to which the operator is applied. If integer together with `shift=True`, the zero-based index of the basis state. Otherwise, directly the quantum number m.
- **shift** (*bool*) – If True for a integer value of *m*, treat *m* as the zero-based index of the basis state (i.e., shift *m* down by s to obtain the quantum number m)

Return type Expr

qnet.algebra.library.spin_algebra.**Jzjmcoeff**(*ls*, *m*, *shift*)
Eigenvalue of the J_z (*Jz*) operator

$$J_z s, m = m s, m$$

See also *Jpjmcoeff()*.

Return type Expr

qnet.algebra.library.spin_algebra.**Jmjmcoeff**(*ls*, *m*, *shift*)
Eigenvalue of the J_- (*Jminus*) operator

$$J_- s, m = \sqrt{s(s+1) - m(m-1)} s, m$$

See also *Jpjmcoeff()*.

Return type Expr

Summary

`__all__` Classes:

Beamsplitter	Infinite bandwidth beamsplitter component.
CoherentDriveCC	Coherent displacement of the input field
Create	Bosonic creation operator
Destroy	Bosonic annihilation operator
Displace	Unitary coherent displacement operator
Jminus	Lowering operator on a spin space
Jplus	Raising operator of a spin space
Jz	Spin (angular momentum) operator in z-direction
Phase	Unitary "phase" operator
PhaseCC	Coherent phase shift cicuit component
SpinOperator	Base class for operators in a spin space
SpinSpace	A Hilbert space for an integer or half-integer spin system
Squeeze	Unitary squeezing operator

`__all__` Functions:

PauliX	Pauli-type X-operator
PauliY	Pauli-type Y-operator
PauliZ	Pauli-type Z-operator
SpinBasisKet	Constructor for a `BasisKet` for a `SpinSpace`

qnet.algebra.pattern_matching package

QNET's pattern matching engine.

Patterns may be constructed by either instantiating a *Pattern* instance directly, or (preferred) by calling the *pattern()*, *pattern_head()*, or *wc()* helper routines.

The pattern may then be matched against an expression using *match_pattern()*. The result of a match is a *MatchDict* object, which evaluates to True or False in a boolean context to indicate the success or failure of the match (or alternatively, through the *success* attribute). The *MatchDict* object also maps any wildcard names to the expression that the corresponding wildcard Pattern matches.

Summary

`__all__` Classes:

MatchDict	Result of a *Pattern.match()*
Pattern	Pattern for matching an expression

Private Classes:

ProtoExpr	Object representing an un-instantiated `Expression`

`__all__` Functions:

match_pattern	Recursively match *expr* with the given *expr_or_pattern*
pattern	'Flat' constructor for the Pattern class
pattern_head	Constructor for a *Pattern* matching a *ProtoExpr*
wc	Constructor for a wildcard-*Pattern*

Reference

class qnet.algebra.pattern_matching.**MatchDict**(**args*)

Bases: `collections.OrderedDict`

Result of a *Pattern.match()*

Dictionary of wildcard names to expressions. Once the value for a key is set, attempting to set it again with a different value raises a `KeyError`. The attribute *merge_lists* may be set to modify this behavior for values that are lists: If it is set to a value different from zero, two lists that are set via the same key are merged. If *merge_lists* is negative, the new values are appended to the existing values; if it is positive, the new values are prepended.

In a boolean context, a *MatchDict* always evaluates as True (even if empty, unlike a normal dictionary), unless the *success* attribute is explicitly set to False (which a failed *Pattern.match()* should do)

Attributes

- **success** (*bool*) – Value of the `MatchDict` object in a boolean context: `bool(match) == match.success`
- **reason** (*str*) – If *success* is False, string explaining why the match failed
- **merge_lists** (*int*) – Code that indicates how to combine multiple values that are lists

update(**others*)
Update dict with entries from *other*

If *other* has an attribute `success=False` and `reason`, those attributes are copied as well

class qnet.algebra.pattern_matching.**Pattern**(*head=None, args=None, kwargs=None, *, mode=1, wc_name=None, conditions=None*)
Bases: `object`

Pattern for matching an expression

Parameters

- **head** (`type or None`) – The type (or tuple of types) of the expression that can be matched. If None, any type of Expression matches
- **args** (`list or None`) – List or tuple of positional arguments of the matched Expression (cf. *Expression.args*). Each element is an expression (to be matched exactly) or another Pattern instance (matched recursively). If None, no arguments are checked
- **kwargs** (`dict or None`) – Dictionary of keyword arguments of the expression (cf. *Expression.kwargs*). As for *args*, each value is an expression or Pattern instance.
- **mode** (`int`) – If the pattern is used to match the arguments of an expression, code to indicate how many arguments the Pattern can consume: *Pattern.single*, *Pattern.one_or_more*, *Pattern.zero_or_more*
- **wc_name** (`str or None`) – If pattern matches an expression, key in the resulting `MatchDict` for the expression. If None, the match will not be recorded in the result
- **conditions** (`list of callables, or None`) – If not None, a list of callables that take *expr* and return a boolean value. If the return value is False, the pattern is determined not to match *expr*.

Note: For (sub-)patterns that occur nested in the *args* attribute of another pattern, only the first or last subpattern may have a *mode* other than *Pattern.single*. This also implies that only one of the *args* may have a *mode* other than *Pattern.single*. This restrictions ensures that patterns can be matched without backtracking, thus guaranteeing numerical efficiency.

Example

Consider the following nested circuit expression:

```
>>> C1 = CircuitSymbol('C1', cdim=3)
>>> C2 = CircuitSymbol('C2', cdim=3)
>>> C3 = CircuitSymbol('C3', cdim=3)
>>> C4 = CircuitSymbol('C4', cdim=3)
>>> perm1 = CPermutation((2, 1, 0))
>>> perm2 = CPermutation((0, 2, 1))
```

(continues on next page)

(continued from previous page)

```
>>> concat_expr = Concatenation(
...                  (C1 << C2 << perm1),
...                  (C3 << C4 << perm2))
```

We may match this with the following pattern:

```
>>> conditions = [lambda c: c.cdim == 3,
...               lambda c: c.label[0] == 'C']
>>> A__Circuit = wc("A__", head=CircuitSymbol,
...                 conditions=conditions)
>>> C__Circuit = wc("C__", head=CircuitSymbol,
...                 conditions=conditions)
>>> B_CPermutation = wc("B", head=CPermutation)
>>> D_CPermutation = wc("D", head=CPermutation)
>>> pattern_concat = pattern(
...         Concatenation,
...         pattern(SeriesProduct, A__Circuit, B_CPermutation),
...         pattern(SeriesProduct, C__Circuit, D_CPermutation))
>>> m = pattern_concat.match(concat_expr)
```

The match returns the following dictionary:

```
>>> result = {'A': [C1, C2], 'B': perm1, 'C': [C3, C4], 'D': perm2}
>>> assert m == result
```

single = 1

one_or_more = 2

zero_or_more = 3

extended_arg_patterns()
Iterator over patterns for positional arguments to be matched

This yields the elements of `args`, extended by their *mode* value

match(*expr*)
Match the given expression (recursively)

Returns a `MatchDict` instance that maps any wildcard names to the expressions that the corresponding wildcard pattern matches. For (sub-)pattern that have a *mode* attribute other than *Pattern.single*, the wildcard name is mapped to a list of all matched expression.

If the match is successful, the resulting `MatchDict` instance will evaluate to True in a boolean context. If the match is not successful, it will evaluate as False, and the reason for failure is available in the *reason* attribute of the `MatchDict` object.

Return type `MatchDict`

findall(*expr*)
list of all matching (sub-)expressions in *expr*

See also:

`finditer()` yields the matches (`MatchDict` instances) for the matched expressions.

finditer(*expr*)
Return an iterator over all matches in *expr*

Iterate over all `MatchDict` results of matches for any matching (sub-)expressions in *expr*. The order of the matches conforms to the equivalent matched expressions returned by `findall()`.

wc_names
Set of all wildcard names occurring in the pattern

`qnet.algebra.pattern_matching.`**`pattern`**(*head*, **args*, *mode=1*, *wc_name=None*, *conditions=None*, ***kwargs*)
'Flat' constructor for the Pattern class

Positional and keyword arguments are mapped into *args* and *kwargs*, respectively. Useful for defining rules that match an instantiated Expression with specific arguments

Return type `Pattern`

`qnet.algebra.pattern_matching.`**`pattern_head`**(**args*, *conditions=None*, *wc_name=None*, ***kwargs*)
Constructor for a `Pattern` matching a `ProtoExpr`

The patterns associated with `_rules` and `_binary_rules` of an `Expression` subclass, or those passed to `Expression.add_rule()`, must be instantiated through this routine. The function does not allow to set a wildcard name (*wc_name* must not be given / be None)

Return type `Pattern`

`qnet.algebra.pattern_matching.`**`wc`**(*name_mode='_'*, *head=None*, *args=None*, *kwargs=None*, ***, *conditions=None*)
Constructor for a wildcard-`Pattern`

Helper function to create a Pattern object with an emphasis on wildcard patterns, if we don't care about the arguments of the matched expressions (otherwise, use `pattern()`)

Parameters

- **name_mode** (`str`) – Combined *wc_name* and *mode* for `Pattern` constructor argument. See below for syntax
- **head** (`type, or None`) – See `Pattern`
- **args** (`list or None`) – See `Pattern`
- **kwargs** (`dict or None`) – See `Pattern`
- **conditions** (`list or None`) – See `Pattern`

The *name_mode* argument uses trailing underscored to indicate the *mode*:

- `A` -> `Pattern(wc_name="A", mode=Pattern.single, ...)`
- `A_` -> `Pattern(wc_name="A", mode=Pattern.single, ...)`
- `B__` -> `Pattern(wc_name="B", mode=Pattern.one_or_more, ...)`
- `B___` -> `Pattern(wc_name="C", mode=Pattern.zero_or_more, ...)`

Return type `Pattern`

`class` `qnet.algebra.pattern_matching.`**`ProtoExpr`**(*args*, *kwargs*, *cls=None*)
Bases: `collections.abc.Sequence`

Object representing an un-instantiated `Expression`

A `ProtoExpr` may be matched by a `Pattern` created via `pattern_head()`. This is used in `Expression.create()`: before an expression is instantiated, a `ProtoExpr` is constructed with the positional and keyword arguments passed to `create()`. Then, this `ProtoExpr` is matched against all the automatic rules `create()` knows about.

Parameters

- **args** (*list*) – positional arguments that would be used in the instantiation of the Expression
- **kwargs** (*dict*) – keyword arguments. Will we converted to an OrderedDict
- **cls** (*class or None*) – The class of the Expression that will ultimately be instantiated.

The combined values of *args* and *kwargs* are accessible as a (mutable) sequence.

instantiate(*cls=None*)
Return an instantiated Expression as `cls.create(*self.args, **self.kwargs)`

Parameters

- **cls** (*class*) – The class of the instantiated expression. If not
- **self.cls will be used.** (*given,*) –

classmethod from_expr(*expr*)
Instantiate proto-expression from the given Expression

`qnet.algebra.pattern_matching.`**match_pattern**(*expr_or_pattern*, *expr*)
Recursively match *expr* with the given *expr_or_pattern*

Parameters

- **expr_or_pattern** (object) – either a direct expression (equal to *expr* for a successful match), or an instance of *Pattern*.
- **expr** (object) – the expression to be matched

Return type *MatchDict*

qnet.algebra.toolbox package

Collection of tools to manually manipulate algebraic expressions

Submodules:

qnet.algebra.toolbox.circuit_manipulation module

Summary

Functions:

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

`__all__`: *connect*

Reference

`qnet.algebra.toolbox.circuit_manipulation.`**connect**(*components*, *connections*, *force_SLH=False*, *expand_simplify=True*)
Connect a list of components according to a list of connections.

Parameters

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

Example

```
>>> A = CircuitSymbol('A', cdim=2)
>>> B = CircuitSymbol('B', cdim=2)
>>> BS = Beamsplitter()
>>> circuit = connect(
...     components=[A, B, BS],
...     connections=[
...         ((A, 0),  (BS, 'in')),
...         ((BS, 'tr'), (B, 0)),
...         ((A, 1), (B, 1))])
>>> print(unicode(circuit).replace('cid(1)', '1'))
(B  1)  Perm(0, 2, 1)  (BS(π/4)  1)  Perm(0, 2, 1)  (A  1)
```

The above example corresponds to the circuit diagram:

```
 ─    ───────     ─
> ├───>         ├───> ├
 │A│    │BS(π/4)│    │B│
> ├─>        ├─> ├
 └─└│  └───────└│  └─
     │           │
─────└───────────└────
```

Raises `ValueError` – if *connections* includes any invalid entries

Note: The list of *components* may contain duplicate entries, but in this case you must use a positional index in *connections* to refer to any duplicate component. Alternatively, use unique components by defining different labels.

---

qnet.algebra.toolbox.commutator_manipulation module

Summary

Functions:

expand_commutators_leibniz	Recursively expand commutators in *expr* according to the Leibniz rule.

`__all__`: *expand_commutators_leibniz*

Reference

qnet.algebra.toolbox.commutator_manipulation.**expand_commutators_leibniz**(*expr*, *expand_expr=True*)

Recursively expand commutators in *expr* according to the Leibniz rule.

$$[AB, C] = A[B, C] + [A, C]B$$

$$[A, BC] = [A, B]C + B[A, C]$$

If *expand_expr* is True, expand products of sums in *expr*, as well as in the result.

qnet.algebra.toolbox.core module

Summary

Functions:

no_instance_caching	Temporarily disable instance caching in *create()*
symbols	The symbols() function from SymPy
temporary_instance_cache	Use a temporary cache for instances in *create()*
temporary_rules	Allow temporary modification of rules for *create()*

`__all__`: *no_instance_caching*, *symbols*, *temporary_instance_cache*, *temporary_rules*

Reference

qnet.algebra.toolbox.core.**no_instance_caching**()

Temporarily disable instance caching in *create()*

Within the managed context, *create()* will not use any caching, for any class.

qnet.algebra.toolbox.core.**temporary_instance_cache**(**classes*)

Use a temporary cache for instances in *create()*

The instance cache used by *create()* for any of the given *classes* will be cleared upon entering the managed context, and restored on leaving it. That is, no cached instances from outside of the managed context will be used within the managed context, and vice versa

qnet.algebra.toolbox.core.**temporary_rules**(**classes*, *clear=False*)

Allow temporary modification of rules for *create()*

For every one of the given *classes*, protect the rules (processed by *match_replace()* or *match_replace_binary()*) associated with that class from modification beyond the managed context. Implies *temporary_instance_cache()*. If *clear* is given as True, all existing rules are temporarily cleared from the given classes on entering the managed context.

Within the managed context, *add_rule()* may be used for any class in *classes* to define local rules, or *del_rules()* to disable specific existing rules (assuming *clear* is False). Upon leaving the managed context all original rules will be restored, removing any local rules.

The *classes*' *simplifications* attribute is also protected from permanent modification. Locally modifying *simplifications* should be done with care, but allows complete control over the creation of expressions.

qnet.algebra.toolbox.core.**symbols**(*names*, ***args*)

The symbols() function from SymPy

This can be used to generate QNET symbols as well:

```
>>> A, B, C = symbols('A B C', cls=OperatorSymbol, hs=0)
>>> srepr(A)
"OperatorSymbol('A', hs=LocalSpace('0'))"
>>> C1, C2 = symbols('C_1:3', cls=CircuitSymbol, cdim=2)
>>> srepr(C1)
"CircuitSymbol('C_1', cdim=2)"
```

Basically, the *cls* keyword argument can be any instantiator, i.e. a class or callable that receives a symbol name as the single positional argument. Any keyword arguments not handled by *symbols()* directly (see sympy.core.symbol.symbols() documentation) is passed on to the instantiator. Obviously, this is extremely flexible.

Note: symbol() does not pass *positional* arguments to the instantiator. Two possible workarounds to create symbols with e.g. a scalar argument are:

```
>>> t = symbols('t', positive=True)
>>> A_t, B_t = symbols(
...     'A B', cls=lambda s: OperatorSymbol(s, t, hs=0))
>>> srepr(A_t, cache={t: 't'})
"OperatorSymbol('A', t, hs=LocalSpace('0'))"
>>> A_t, B_t = (OperatorSymbol(s, t, hs=0) for s in ('A', 'B'))
>>> srepr(B_t, cache={t: 't'})
"OperatorSymbol('B', t, hs=LocalSpace('0'))"
```

qnet.algebra.toolbox.equation module

Tools for working with equations

Summary

Classes:

Eq	Symbolic equation

__all__: *Eq*

Reference

class qnet.algebra.toolbox.equation.**Eq**(*lhs*, *rhs*, *tag=None*, *_prev_lhs=None*, *_prev_rhs=None*, *_prev_tags=None*)

Bases: object

Symbolic equation

This class keeps track of the *lhs* and *rhs* of an equation across arbitrary manipulations

Parameters

- **lhs** (Expression) – the left-hand-side of the equation
- **rhs** (Expression) – the right-hand-side of the equation
- **tag** (*None or str*) – a tag (equation number) to be shown when printing the equation

Example

```
>>> ω, E0 = sympy.symbols('omega, E_0')
>>> hbar = sympy.symbols('hbar', positive=True)
>>> H_0, H_1 = (OperatorSymbol(s, hs=0) for s in ('H_0', 'H_1'))
>>> H = OperatorSymbol('H', hs=0)
>>> mu = OperatorSymbol('mu', hs=0)
>>> eq0 = Eq(H_0, ω * Create(hs=0) * Destroy(hs=0) + E0, tag='0')
>>> print(unicode(eq0, show_hs_label=False))
H₀ = E₀ + ω a^† a    (0)
>>> eq1 = Eq(H_1, mu + E0, tag='1')
>>> print(unicode(eq1, show_hs_label=False))
H₁ = E₀ + μ    (1)
>>> eq = (
...     (eq0 + eq1).set_tag('2')
...     .apply_to_rhs(lambda expr: expr - 2*E0, cont=True)
...     .apply(lambda expr: expr * hbar, cont=True)
...     .apply_mtd_to_lhs(
...         'substitute', var_map={H_0 + H_1: H}, cont=True)
...     .apply(lambda expr: expr**2, cont=True)
...     .apply_mtd_to_rhs('substitute', var_map={mu: 0}, cont=True)
...     .apply_mtd_to_rhs('expand', cont=True, tag='')
... )
>>> print(unicode(eq, show_hs_label=False))
    H₀ + H₁ = 2 E₀ + μ + ω a^† a                  (2)
            = μ + ω a^† a
h (H₀ + H₁) = h (μ + ω a^† a)
        h H = h (μ + ω a^† a)
     h² H H = h² (μ + ω a^† a) (μ + ω a^† a)
            = h² ω² a^† ( + a^† a) a
            = h² ω² a^† a^† a a + h² ω² a^† a     ()
>>> (eq
...  .apply_mtd_to_lhs('substitute', eq.as_dict)
...  .verify().is_zero)
True
```

lhs
The left-hand-side of the equation

rhs
The right-hand-side of the equation

tag
A tag (equation number) to be shown when printing the equation, or None

set_tag(*tag*)
Return a copy of the equation with a new *tag*

as_dict
Mapping of the lhs to the rhs

This allows to plug an equation into another expression via *substitute()*.

apply (*func*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Apply *func* to both sides of the equation

Returns a new equation where the left-hand-side and right-hand side are replaced by the application of *func*:

```
lhs=func(lhs, *args, **kwargs)
rhs=func(rhs, *args, **kwargs)
```

If `cont=True`, the resulting equation will keep a history of its previous state (resulting in multiple lines of equations when printed, as in the main example above).

The resulting equation with have the given *tag*.

apply_to_lhs (*func*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Apply *func* to lhs of equation only

Like `apply()`, but modifying only the left-hand-side.

apply_to_rhs (*func*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Apply *func* to rhs of equation only

Like `apply()`, but modifying only the right-hand-side.

apply_mtd (*mtd*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Call the method *mtd* on both sides of the equation

That is, the left-hand-side and right-hand-side are replaced by:

```
lhs=lhs.<mtd>(*args, **kwargs)
rhs=rhs.<mtd>(*args, **kwargs)
```

The *cont* and *tag* parameters are as in `apply()`.

apply_mtd_to_lhs (*mtd*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Call the method *mtd* on the lhs of the equation only.

Like `apply_mtd()`, but modifying only the left-hand-side.

apply_mtd_to_rhs (*mtd*, **args*, *cont=False*, *tag=None*, ***kwargs*)
Call the method *mtd* on the rhs of the equation

Like `apply_mtd()`, but modifying only the right-hand-side.

substitute (*var_map*, *cont=False*, *tag=None*)
Substitute sub-expressions both on the lhs and rhs

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

verify (*func=None*, **args*, ***kwargs*)
Subtract the rhs from the lhs of the equation

Before the substraction, each side is expanded and any scalars are simplified. If given, *func* with the positional arguments *args* and keyword-arguments *kwargs* is applied to the result before returning it.

You may complete the verification by checking the `is_zero` attribute of the returned expression.

copy ()
Return a copy of the equation

free_symbols
Set of free SymPy symbols contained within the equation.

bound_symbols
Set of bound SymPy symbols contained within the equation.

all_symbols
Combination of `free_symbols` and `bound_symbols`

Summary

`__all__` Classes:

`Eq`	Symbolic equation

`__all__` Functions:

`connect`	Connect a list of components according to a list of connections.
`expand_commutators_leibniz`	Recursively expand commutators in *expr* according to the Leibniz rule.
`no_instance_caching`	Temporarily disable instance caching in `create()`
`symbols`	The `symbols()` function from SymPy
`temporary_instance_cache`	Use a temporary cache for instances in `create()`
`temporary_rules`	Allow temporary modification of rules for `create()`

Summary

`__all__` Exceptions:

`AlgebraError`	Base class for all algebraic errors
`AlgebraException`	Base class for all algebraic exceptions
`BadLiouvillianError`	Raised when a Liouvillian is not of standard Lindblad form.
`BasisNotSetError`	Raised if the basis or a Hilbert space dimension is unavailable
`CannotConvertToSLH`	Raised when a circuit algebra object cannot be converted to SLH
`CannotEliminateAutomatically`	Raised when attempted automatic adiabatic elimination fails.
`CannotSimplify`	Raised when a rule cannot further simplify an expression
`CannotSymbolicallyDiagonalize`	Matrix cannot be diagonalized analytically.
`CannotVisualize`	Raised when a circuit cannot be visually represented.
`IncompatibleBlockStructures`	Raised for invalid block-decomposition
`InfiniteSumError`	Raised when expanding a sum into an infinite number of terms
`NoConjugateMatrix`	Raised when entries of `Matrix` have no defined conjugate
`NonSquareMatrix`	Raised when a `Matrix` fails to be square
`OverlappingSpaces`	Raised when objects fail to be in separate Hilbert spaces.
`SpaceTooLargeError`	Raised when objects fail to be have overlapping Hilbert spaces.
`UnequalSpaces`	Raised when objects fail to be in the same Hilbert space.
`WrongCDimError`	Raised for mismatched channel number in circuit series

`__all__` Classes:

`Adjoint`	Symbolic Adjoint of an operator
`BasisKet`	Local basis state, identified by index or label
`Beamsplitter`	Infinite bandwidth beamsplitter component.
`Bra`	The associated dual/adjoint state for any ket

Continued on next page

Table 37 – continued from previous page

BraKet	The symbolic inner product between two states
CPermutation	Channel permuting circuit
Circuit	Base class for the circuit algebra elements
CircuitSymbol	Symbolic circuit element
CoherentDriveCC	Coherent displacement of the input field
CoherentStateKet	Local coherent state, labeled by a complex amplitude
Commutator	Commutator of two operators
Component	Base class for circuit components
Concatenation	Concatenation of circuit elements
Create	Bosonic creation operator
Destroy	Bosonic annihilation operator
Displace	Unitary coherent displacement operator
Eq	Symbolic equation
Expression	Base class for all QNET Expressions
Feedback	Feedback on a single channel of a circuit
HilbertSpace	Base class for Hilbert spaces
IndexedSum	Base class for indexed sums
Jminus	Lowering operator on a spin space
Jplus	Raising operator of a spin space
Jz	Spin (angular momentum) operator in z-direction
KetBra	Outer product of two states
KetIndexedSum	Indexed sum over Kets
KetPlus	Sum of states
KetSymbol	Symbolic state
LocalKet	A state on a `LocalSpace`
LocalOperator	Base class for "known" operators on a `LocalSpace`
LocalSigma	Level flip operator between two levels of a `LocalSpace`
LocalSpace	Hilbert space for a single degree of freedom.
MatchDict	Result of a `Pattern.match()`
Matrix	Matrix of Expressions
NullSpaceProjector	Projection operator onto the nullspace of its operand
Operation	Base class for "operations"
Operator	Base class for all quantum operators.
OperatorDerivative	Symbolic partial derivative of an operator
OperatorIndexedSum	Indexed sum over operators
OperatorPlus	Sum of Operators
OperatorPlusMinusCC	An operator plus or minus its complex conjugate
OperatorSymbol	Symbolic operator
OperatorTimes	Product of operators
OperatorTimesKet	Product of an operator and a state.
OperatorTrace	(Partial) trace of an operator
Pattern	Pattern for matching an expression
Phase	Unitary "phase" operator
PhaseCC	Coherent phase shift cicuit component
ProductSpace	Tensor product of local Hilbert spaces
PseudoInverse	Unevaluated pseudo-inverse X^+ of an operator X
QuantumAdjoint	Base class for adjoints of quantum expressions
QuantumDerivative	Symbolic partial derivative
QuantumExpression	Base class for expressions associated with a Hilbert space
QuantumIndexedSum	Base class for indexed sums

Continued on next page

Table 37 – continued from previous page

QuantumOperation	Base class for operations on quantum expression
QuantumPlus	General implementation of addition of quantum expressions
QuantumSymbol	Symbolic element of an algebra
QuantumTimes	General implementation of product of quantum expressions
SLH	Element of the SLH algebra
SPost	Linear post-multiplication operator
SPre	Linear pre-multiplication operator
Scalar	Base class for Scalars
ScalarDerivative	Symbolic partial derivative of a scalar
ScalarExpression	Base class for scalars with non-scalar arguments
ScalarIndexedSum	Indexed sum over scalars
ScalarPlus	Sum of scalars
ScalarPower	A scalar raised to a power
ScalarTimes	Product of scalars
ScalarTimesKet	Product of a Scalar coefficient and a ket
ScalarTimesOperator	Product of a Scalar coefficient and an Operator
ScalarTimesQuantumExpression	Product of a Scalar and a QuantumExpression
ScalarTimesSuperOperator	Product of a Scalar coefficient and a SuperOperator
ScalarValue	Wrapper around a numeric or symbolic value
SeriesInverse	Symbolic series product inversion operation
SeriesProduct	The series product circuit operation.
SingleQuantumOperation	Base class for operations on a single quantum expression
SpinOperator	Base class for operators in a spin space
SpinSpace	A Hilbert space for an integer or half-integer spin system
Squeeze	Unitary squeezing operator
State	Base class for states in a Hilbert space
StateDerivative	Symbolic partial derivative of a state
SuperAdjoint	Adjoint of a super-operator
SuperOperator	Base class for super-operators
SuperOperatorDerivative	Symbolic partial derivative of a super-operator
SuperOperatorPlus	A sum of super-operators
SuperOperatorSymbol	Symbolic super-operator
SuperOperatorTimes	Product of super-operators
SuperOperatorTimesOperator	Application of a super-operator to an operator
TensorKet	A tensor product of kets

`__all__` Functions:

FB	Wrapper for Feedback, defaulting to last channel
KroneckerDelta	Kronecker delta symbol
LocalProjector	A projector onto a specific level of a LocalSpace
PauliX	Pauli-type X-operator
PauliY	Pauli-type Y-operator
PauliZ	Pauli-type Z-operator
SpinBasisKet	Constructor for a BasisKet for a SpinSpace
Sum	Instantiator for an arbitrary indexed sum.
adjoint	Return the adjoint of an obj.
anti_commutator	If B != None, return the anti-commutator $\{A, B\}$, otherwise return the super-operator {
block_matrix	Generate the operator matrix with quadrants
circuit_identity	Return the circuit identity for n channels

commutator	Commutator of *A* and *B*
connect	Connect a list of components according to a list of connections.
decompose_space	Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated
diagm	Generalizes the diagonal matrix creation capabilities of *numpy.diag* to Matrix objects.
eval_adiabatic_limit	Compute the limiting SLH model for the adiabatic approximation
expand_commutators_leibniz	Recursively expand commutators in *expr* according to the Leibniz rule.
extract_channel	Create a CPermutation that extracts channel *k*
factor_coeff	Factor out coefficients of all factors.
factor_for_trace	Given a LocalSpace *ls* to take the partial trace over and an operator *op*, factor the trace
getABCD	Calculate the ABCD-linearization of an SLH model
get_coeffs	Create a dictionary with all Operator terms of the expression (understood as a sum) as keys
hstackm	Generalizes *numpy.hstack* to Matrix objects.
identity_matrix	Generate the N-dimensional identity matrix.
init_algebra	Initialize the algebra system
lindblad	Return the super-operator Lindblad term of the Lindblad operator *C*
liouvillian	Return the Liouvillian super-operator associated with *H* and *Ls*
liouvillian_normal_form	Return a Hamilton operator H and a minimal list of collapse operators Ls that generate the
map_channels	Create a CPermuation based on a dict of channel mappings
match_pattern	Recursively match *expr* with the given *expr_or_pattern*
move_drive_to_H	Move coherent drives from the Lindblad operators to the Hamiltonian.
no_instance_caching	Temporarily disable instance caching in create()
pad_with_identity	Pad a circuit by adding a *n*-channel identity circuit at index *k*
pattern	'Flat' constructor for the Pattern class
pattern_head	Constructor for a Pattern matching a ProtoExpr
prepare_adiabatic_limit	Prepare the adiabatic elimination on an SLH object
rewrite_with_operator_pm_cc	Try to rewrite expr using OperatorPlusMinusCC
sqrt	Square root of a Scalar or scalar value
substitute	Substitute symbols or (sub-)expressions with the given replacements and re-evalute the res
symbols	The symbols() function from SymPy
temporary_instance_cache	Use a temporary cache for instances in create()
temporary_rules	Allow temporary modification of rules for create()
try_adiabatic_elimination	Attempt to automatically do adiabatic elimination on an SLH object
vstackm	Generalizes *numpy.vstack* to Matrix objects.
wc	Constructor for a wildcard-Pattern
zerosm	Generalizes numpy.zeros to Matrix objects.

__all__ Data:

CIdentity	Single pass-through channel; neutral element of SeriesProduct
CircuitZero	Zero circuit, the neutral element of Concatenation
FullSpace	The 'full space', i.e.
II	IdentityOperator constant (singleton) object.
IdentityOperator	IdentityOperator constant (singleton) object.
IdentitySuperOperator	Neutral element for product of super-operators
One	The neutral element with respect to scalar multiplication
TrivialKet	TrivialKet constant (singleton) object.
TrivialSpace	The 'nullspace', i.e.
Zero	The neutral element with respect to scalar addition
ZeroKet	ZeroKet constant (singleton) object for the null-state.
ZeroOperator	ZeroOperator constant (singleton) object.
ZeroSuperOperator	Neutral element for sum of super-operators
tr	Instantiate while applying automatic simplifications

Reference

qnet.algebra.**init_algebra**(*, *default_hs_cls='LocalSpace'*)
Initialize the algebra system

Parameters **default_hs_cls** (*str*) – The name of the *LocalSpace* subclass that should be used when implicitly creating Hilbert spaces, e.g. in *OperatorSymbol*

9.1.2 qnet.convert package

Conversion to QuTiP and Sympy

Submodules:

qnet.convert.to_qutip module

Conversion of QNET expressions to qutip objects.

Summary

Functions:

SLH_to_qutip	Generate and return QuTiP representation matrices for the Hamiltonian and the collapse operators.
convert_to_qutip	Convert a QNET expression to a qutip object

__all__: *SLH_to_qutip*, *convert_to_qutip*

Reference

qnet.convert.to_qutip.**convert_to_qutip**(*expr*, *full_space=None*, *mapping=None*)
Convert a QNET expression to a qutip object

Parameters

- **expr** – a QNET expression
- **full_space** (HilbertSpace) – The Hilbert space in which *expr* is defined. If not given, expr.space is used. The Hilbert space must have a well-defined basis.
- **mapping** (*dict*) – A mapping of any (sub-)expression to either a *quip.Qobj* directly, or to a callable that will convert the expression into a *qutip.Qobj*. Useful for e.g. supplying objects for symbols

Raises ValueError – if *expr* is not in *full_space*, or if *expr* cannot be converted.

qnet.convert.to_qutip.**SLH_to_qutip**(*slh*, *full_space=None*, *time_symbol=None*, *convert_as='pyfunc'*)

Generate and return QuTiP representation matrices for the Hamiltonian and the collapse operators. Any inhomogeneities in the Lindblad operators (resulting from coherent drives) will be moved into the Hamiltonian, cf. move_drive_to_H().

Parameters

- **slh** (SLH) – The SLH object from which to generate the qutip data
- **full_space** (HilbertSpace *or* None) – The Hilbert space in which to represent the operators. If None, the space of *shl* will be used
- **time_symbol** (sympy.Symbol or None) – The symbol (if any) expressing time dependence (usually 't')
- **convert_as** (*str*) – How to express time dependencies to qutip. Must be 'pyfunc' or 'str'

Returns tuple (H, [L1, L2, ...]) as numerical *qutip.Qobj* representations, where H and each L may be a nested list to express time dependence, e.g. H = [H0, [H1, eps_t]], where H0 and H1 are of type *qutip.Qobj*, and eps_t is either a string (convert_as='str') or a function (convert_as='pyfunc')

Raises AlgebraError – If the Hilbert space (*slh.space* or *full_space*) is invalid for numerical conversion

qnet.convert.to_sympy_matrix module

Conversion of QNET expressions to sympy matrices. For small Hilbert spaces, this facilitates some analytic treatments, such as decomposition into a basis.

Summary

Functions:

SympyCreate	Creation operator for a Hilbert space of dimension *n*, as an instance of *sympy.Matrix*
basis_state	n x 1 *sympy.Matrix* representing the *i*'th eigenstate of an *n*-dimensional Hilbert space (*i* >= 0)
convert_to_sympy_matrix	Convert a QNET expression to an explicit n x n instance of *sympy.Matrix*, where n is the dimension of *full_space*.

__all__: *convert_to_sympy_matrix*

Reference

qnet.convert.to_sympy_matrix.**basis_state**(*i*, *n*)
: `n x 1` *sympy.Matrix* representing the *i*'th eigenstate of an *n*-dimensional Hilbert space (*i* >= 0)

qnet.convert.to_sympy_matrix.**SympyCreate**(*n*)
: Creation operator for a Hilbert space of dimension *n*, as an instance of *sympy.Matrix*

qnet.convert.to_sympy_matrix.**convert_to_sympy_matrix**(*expr*, *full_space=None*)
: Convert a QNET expression to an explicit `n x n` instance of *sympy.Matrix*, where `n` is the dimension of *full_space*. The entries of the matrix may contain symbols.

Parameters

- **expr** – a QNET expression
- **full_space** (*qnet.algebra.hilbert_space_algebra.HilbertSpace*) – The Hilbert space in which *expr* is defined. If not given, `expr.space` is used. The Hilbert space must have a well-defined basis.

Raises

- `qnet.algebra.hilbert_space_algebra.BasisNotSetError` – if *full_space* does not have a defined basis
- `ValueError` – if *expr* is not in *full_space*, or if *expr* cannot be converted.

Summary

`__all__` Functions:

SLH_to_qutip	Generate and return QuTiP representation matrices for the Hamiltonian and the collapse operators.
convert_to_qutip	Convert a QNET expression to a qutip object
convert_to_sympy_matrix	Convert a QNET expression to an explicit `n x n` instance of *sympy.Matrix*, where `n` is the dimension of *full_space*.

9.1.3 qnet.printing package

Printing system for QNET Expressions and related objects

Submodules:

qnet.printing.asciiprinter module

ASCII Printer

Summary

Classes:

QnetAsciiDefaultPrinter	Printer for an ASCII representation that accepts no settings.

Continued on next page

Table 41 – continued from previous page

QnetAsciiPrinter	Printer for a string (ASCII) representation.

Reference

class qnet.printing.asciiprinter.**QnetAsciiPrinter**(*cache=None*, *settings=None*)

Bases: *qnet.printing.base.QnetBasePrinter*

Printer for a string (ASCII) representation.

Attributes

- **_parenth_left** (*str*) – String to use for a left parenthesis (e.g. 'left(' in LaTeX). Used by `_split_op()`
- **_parenth_left** (*str*) – String to use for a right parenthesis
- **_dagger_sym** (*str*) – Symbol that indicates the complex conjugate of an operator. Used by `_split_op()`
- **_tensor_sym** (*str*) – Symbol to use for tensor products. Used by `_render_hs_label()`.

sympy_printer_cls

alias of *qnet.printing.sympy.SympyStrPrinter*

printmethod = '_ascii'

parenthesize(*expr*, *level*, **args*, *strict=False*, ***kwargs*)

Render *expr* and wrap the result in parentheses if the precedence of *expr* is below the given *level* (or at the given *level* if *strict* is True. Extra *args* and *kwargs* are passed to the internal *doit* renderer

class qnet.printing.asciiprinter.**QnetAsciiDefaultPrinter**

Bases: *qnet.printing.asciiprinter.QnetAsciiPrinter*

Printer for an ASCII representation that accepts no settings. This can be used internally when a well-defined, static representation is needed (e.g. as a sort key)

qnet.printing.base module

Provides the base class for Printers

Summary

Classes:

QnetBasePrinter	Base class for all QNET expression printers

Reference

class qnet.printing.base.**QnetBasePrinter**(*cache=None*, *settings=None*)

Bases: `sympy.printing.printer.Printer`

Base class for all QNET expression printers

Parameters

- **cache** (*dict or None*) – A dict that maps expressions to strings. It may be given during istantiation to use pre-defined strings for specific expressions. The cache will be updated as the printer is used.
- **settings** (*dict or None*) – A dict of settings.

Class Attributes

- **sympy_printer_cls** (*type*) – The class that will be instantiated to print Sympy expressions
- **_default_settings** (*dict*) – The default value of all settings. Note only settings for which there are defaults defined here are accepted when instantiating the printer
- **printmethod** (*None or str*) – Name of a method that expressions may define to print themeselves.

Attributes

- **cache** (*dict*) – Dictionary where the results of any call to `doprint()` is stored. When `doprint()` is called for an expression that is already in *cache*, the result from the cache is returned.
- **_sympy_printer** (*sympy.printing.printer.Printer*) – The printer instance that will be used to print any Sympy expression.
- **_allow_caching** (*bool*) – A flag that may be set to completely disable caching
- **_print_level** (*int*) – The recursion depth of `doprint()` (>= 1 inside any of the `_print*` methods)

Raises `TypeError` – If any key in *settings* is not defined in the *_default_settings* of the printer, respectively the *sympy_printer_cls*.

sympy_printer_cls
alias of `qnet.printing.sympy.SympyStrPrinter`

printmethod = None

emptyPrinter(*expr*)
Fallback method for expressions that neither know how to print themeselves, nor for which the printer has a suitable `_print*` method

doprint(*expr*, **args*, ***kwargs*)
Returns printer's representation for expr (as a string)

The representation is obtained by the following methods:

1. from the `cache`
2. If *expr* is a Sympy object, delegate to the `doprint()` method of `_sympy_printer`
3. Let the *expr* print itself if has the `printmethod`
4. Take the best fitting `_print_*` method of the printer
5. As fallback, delegate to `emptyPrinter()`

Any extra *args* or *kwargs* are passed to the internal *_print* method.

qnet.printing.dot module

DOT printer for Expressions.

This module provides the `dotprint()` function that generates a DOT diagram for a given expression. For example:

```
>>> A = OperatorSymbol("A", hs=1)
>>> B = OperatorSymbol("B", hs=1)
>>> expr = 2 * (A + B)
>>> with configure_printing(str_format='unicode'):
...     dot = dotprint(expr)
>>> dot.strip() == r'''
... digraph{
...
... # Graph style
... "ordering"="out"
... "rankdir"="TD"
...
... #########
... # Nodes #
... #########
...
... "node_(0, 0)" ["label"="ScalarTimesOperator"];
... "node_(1, 0)" ["label"="2"];
... "node_(1, 1)" ["label"="OperatorPlus"];
... "node_(2, 0)" ["label"="Â¹"];
... "node_(2, 1)" ["label"="B̂¹"];
...
... #########
... # Edges #
... #########
...
... "node_(0, 0)" -> "node_(1, 0)"
... "node_(0, 0)" -> "node_(1, 1)"
... "node_(1, 1)" -> "node_(2, 0)"
... "node_(1, 1)" -> "node_(2, 1)"
... }'''.strip()
True
```

The `dot` commandline program renders the code into an image:

The various options of *dotprint()* allow for arbitrary customization of the graph's structural and visual properties.

Summary

Functions:

dotprint	Return the DOT (graph) description of an Expression tree as a string
expr_labelfunc	Factory for function `labelfunc(expr, is_leaf)`

Reference

`qnet.printing.dot.`**`expr_labelfunc`**(*leaf_renderer=<class 'str'>*, *fallback=<class 'str'>*)

Factory for function `labelfunc(expr, is_leaf)`

It has the following behavior:

- If `is_leaf` is True, return `leaf_renderer(expr)`.
- Otherwise,
  - if *expr* is an Expression, return a custom string similar to `srepr()`, but with an ellipsis for `args`
  - otherwise, return `fallback(expr)`

`qnet.printing.dot.dotprint`(*expr*, *styles=None*, *maxdepth=None*, *repeat=True*, *labelfunc=<function expr_labelfunc.<locals>._labelfunc>*, *idfunc=None*, *get_children=<function _op_children>*, ***kwargs*)

Return the DOT (graph) description of an Expression tree as a string

Parameters

- **expr** (*object*) – The expression to render into a graph. Typically an instance of `Expression`, but with appropriate *get_children*, *labelfunc*, and *id_func*, this could be any tree-like object
- **styles** (*list or None*) – A list of tuples `(expr_filter, style_dict)` where `expr_filter` is a callable and `style_dict` is a list of DOT node properties that should be used when rendering a node for which `expr_filter(expr)` return True.
- **maxdepth** (*int or None*) – The maximum depth of the resulting tree (any node at *maxdepth* will be drawn as a leaf)
- **repeat** (*bool*) – By default, if identical sub-expressions occur in multiple locations (as identified by *idfunc*, they will be repeated in the graph. If `repeat=False` is given, each unique (sub-)expression is only drawn once. The resulting graph may no longer be a proper tree, as recurring expressions will have multiple parents.
- **labelfunc** (*callable*) – A function that receives *expr* and a boolean `is_leaf` and returns the label of the corresponding node in the graph. Defaults to `expr_labelfunc(str, str)`.
- **idfunc** (*callable or None*) – A function that returns the ID of the node representing a given expression. Expressions for which *idfunc* returns identical results are considered identical if *repeat* is False. The default value None uses a function that is appropriate to a single standalone DOT file. If this is insufficient, something like `hash` or `str` would make a good *idfunc*.
- **get_children** (*callable*) – A function that return a list of sub-expressions (the children of *expr*). Defaults to the operands of an `Operation` (thus, anything that is not an Operation is a leaf)
- **kwargs** – All further keyword arguments set custom DOT graph attributes

Returns a multiline str representing a graph in the DOT language

Return type str

Notes

The node *styles* are additive. For example, consider the following custom styles:

```
styles = [
    (lambda expr: isinstance(expr, SCALAR_TYPES),
        {'color': 'blue', 'shape': 'box', 'fontsize': 12}),
    (lambda expr: isinstance(expr, Expression),
        {'color': 'red', 'shape': 'box', 'fontsize': 12}),
```

(continues on next page)

(continued from previous page)

```
    (lambda expr: isinstance(expr, Operation),
        {'color': 'black', 'shape': 'ellipse'})]
```

For Operations (which are a subclass of Expression) the color and shape are overwritten, while the fontsize 12 is inherited.

Keyword arguments are directly translated into graph styles. For example, in order to produce a horizontal instead of vertical graph, use `dotprint(..., rankdir='LR')`.

See also:

`sympy.printing.dot.dotprint()` provides an equivalent function for SymPy expressions.

qnet.printing.latexprinter module

Routines for rendering expressions to LaTeX

Summary

Classes:

QnetLatexPrinter	Printer for a LaTeX representation.

Functions:

render_latex_sub_super	Assemble a string from the primary name and the given sub- and superscripts.

Reference

class qnet.printing.latexprinter.**QnetLatexPrinter**(*cache=None*, *settings=None*)

Bases: *qnet.printing.asciiprinter.QnetAsciiPrinter*

Printer for a LaTeX representation.

See *qnet.printing.latex()* for documentation of *settings*.

sympy_printer_cls

alias of *qnet.printing.sympy.SympyLatexPrinter*

printmethod = '_latex'

qnet.printing.latexprinter.**render_latex_sub_super**(*name*, *subs=None*, *supers=None*, *translate_symbols=True*, *sep=', '*)

Assemble a string from the primary name and the given sub- and superscripts:

```
>>> render_latex_sub_super(name='alpha', subs=['mu', 'nu'], supers=[2])
'\\alpha_{\\mu,\\nu}^{2}'

>>> render_latex_sub_super(
...     name='alpha', subs=['1', '2'], supers=['(1)'], sep='')
'\\alpha_{12}^{(1)}'
```

Parameters

- **name** (*str*) – the string without the subscript/superscript
- **subs** (*list or None*) – list of subscripts
- **supers** (*list or None*) – list of superscripts
- **translate_symbols** (*bool*) – If True, try to translate (Greek) symbols in *name, 'subs*, and *supers* to unicode
- **sep** (*str*) – Separator to use if there are multiple subscripts/superscripts

qnet.printing.sreprprinter module

Provides printers for a full-structured representation

Summary

Classes:

IndentedSReprPrinter	Printer for rendering an expression in such a way that the resulting string can be evaluated in an appropriate context to re-instantiate an identical object, using nested indentation (implementing `srepr(expr, indented=True)`
IndentedSympyReprPrinter	Indented repr printer for Sympy objects
QnetSReprPrinter	Printer for a string (ASCII) representation.

Reference

class qnet.printing.sreprprinter.**QnetSReprPrinter**(*cache=None*, *settings=None*)
Bases: *qnet.printing.base.QnetBasePrinter*

Printer for a string (ASCII) representation.

sympy_printer_cls
alias of *qnet.printing.sympy.SympyReprPrinter*

emptyPrinter(*expr*)
Fallback printer

class qnet.printing.sreprprinter.**IndentedSympyReprPrinter**(*settings=None*)
Bases: *qnet.printing.sympy.SympyReprPrinter*

Indented repr printer for Sympy objects

doprint(*expr*)
Returns printer's representation for expr (as a string)

class qnet.printing.sreprprinter.**IndentedSReprPrinter**(*cache=None*, *settings=None*)
Bases: *qnet.printing.base.QnetBasePrinter*

Printer for rendering an expression in such a way that the resulting string can be evaluated in an appropriate context to re-instantiate an identical object, using nested indentation (implementing `srepr(expr, indented=True)`

sympy_printer_cls
alias of *IndentedSympyReprPrinter*

emptyPrinter(*expr*)
Fallback printer

qnet.printing.sympy module

Custom Printers for Sympy expressions

These classes are used by default by the QNET printing systems as sub-printers for SymPy objects (e.g. for symbolic coefficients). They fix some issues with SymPy's builtin printers:

- factors like $\frac{1}{\sqrt{2}}$ occur very commonly in quantum mechanics, and it is standard notation to write them as such. SymPy insists on rationalizing denominators, using $\frac{\sqrt{2}}{2}$ instead. Our custom printers restore the canonical form. Note that internally, Sympy still uses the rationalized structure; but in any case, Sympy makes no guarantees between the algebraic structure of an expression and how it is printed.
- Symbols (especially greek letters) are extremely common, and it's much more readable if the string representation of an expression uses unicode for these. SymPy supports unicode "pretty-printing" (`sympy.printing.pretty.pretty.pretty_print()`) only in "2D", where expressions are rendered as multiline unicode strings. While this is fine for interactive display, it does not work so well for a simple `str`. The *SympyUnicodePrinter* solves this by producing simple strings with unicode symbols.
- Some algebraic structures such as factorials, complex-conjugates and indexed symbols have sub-optimal rendering in `sympy.printing.str.StrPrinter`
- QNET contains some custom subclasses of SymPy objects (e.g. *IdxSym*) that the default printers don't know how to deal with (respectively, render incorrectly!)

Summary

Classes:

SympyLatexPrinter	Variation of sympy LatexPrinter that derationalizes denominators
SympyReprPrinter	Representation printer with support for *IdxSym*
SympyStrPrinter	Variation of sympy StrPrinter that derationalizes denominators.
SympyUnicodePrinter	Printer that represents SymPy expressions as (single-line) unicode strings.

Functions:

derationalize_denom	Try to de-rationalize the denominator of the given expression.

Reference

qnet.printing.sympy.**derationalize_denom**(*expr*)
Try to de-rationalize the denominator of the given expression.

The purpose is to allow to reconstruct e.g. `1/sqrt(2)` from `sqrt(2)/2`.

Specifically, this matches *expr* against the following pattern:

```
Mul(..., Rational(n, d), Pow(d, Rational(1, 2)), ...)
```

and returns a tuple `(numerator, denom_sq, post_factor)`, where `numerator` and `denom_sq` are `n` and `d` in the above pattern (of type *int*), respectively, and `post_factor` is the product of the remaining factors (`...` in *expr*). The result will fulfill the following identity:

```
(numerator / sqrt(denom_sq)) * post_factor == expr
```

If *expr* does not follow the appropriate pattern, a `ValueError` is raised.

class `qnet.printing.sympy.`**`SympyStrPrinter`**`(`*settings=None*`)`
Bases: `sympy.printing.str.StrPrinter`

Variation of sympy StrPrinter that derationalizes denominators.

Additionally, it contains the following modifications:

- Support for *`IdxSym`*
- Rendering of `sympy.tensor.indexed.Indexed` as subscripts
- Rendering of `sympy.functions.combinatorial.factorials.factorial` as `!`
- Option *conjg_style* to configure how complex conjugates are rendered: `'func' renders it as ``conjugate(...)`, and `'star'` uses an exponentiated asterisk

printmethod = '_sympystr'

class `qnet.printing.sympy.`**`SympyLatexPrinter`**`(`*settings=None*`)`
Bases: `sympy.printing.latex.LatexPrinter`

Variation of sympy LatexPrinter that derationalizes denominators

Additionally, it contains the following modifications:

- Support for *`IdxSym`*
- A setting *conjg_style* that allows to specify how complex conjugate are rendered: `'overline'` (the default) draws a line over the number, 'star' uses an exponentiated asterisk, and 'func' renders a a `conjugate` function

printmethod = '_latex'

class `qnet.printing.sympy.`**`SympyUnicodePrinter`**`(`*settings=None*`)`
Bases: *`qnet.printing.sympy.SympyStrPrinter`*

Printer that represents SymPy expressions as (single-line) unicode strings.

This is a mixture of `StrPrinter` and `sympy.printing.pretty.pretty.PrettyPrinter` (minus the 2D printing), with the same extensions as *`SympyStrPrinter`*

printmethod = '_sympystr'

class `qnet.printing.sympy.`**`SympyReprPrinter`**`(`*settings=None*`)`
Bases: `sympy.printing.repr.ReprPrinter`

Representation printer with support for *`IdxSym`*

qnet.printing.treeprinting module

Tree printer for Expressions

This is mainly for interactive use.

Summary

Functions:

print_tree	Print a tree representation of the structure of *expr*
tree	Give the output of *tree* as a multiline string, using line drawings to visualize the hierarchy of expressions (similar to the `tree` unix command line program for showing directory trees)

Reference

`qnet.printing.treeprinting.`**`print_tree`**(*expr*, *attr='operands'*, *padding=''*, *exclude_type=None*, *depth=None*, *unicode=True*, *srepr_leaves=False*, *_last=False*, *_root=True*, *_level=0*, *_print=True*)

Print a tree representation of the structure of *expr*

Parameters

- **expr** (`Expression`) – expression to render
- **attr** (*str*) – The attribute from which to get the children of *expr*
- **padding** (*str*) – Whitespace by which the entire tree is idented
- **exclude_type** (*type*) – Type (or list of types) which should never be expanded recursively
- **depth** (*int or None*) – Maximum depth of the tree to be printed
- **unicode** (*bool*) – If True, use unicode line-drawing symbols for the tree, and print expressions in a unicode representation. If False, use an ASCII approximation.
- **srepr_leaves** (*bool*) – Whether or not to render leaves with *srepr*, instead of *ascii/unicode*

See also:

tree() return the result as a string, instead of printing it

`qnet.printing.treeprinting.`**`tree`**(*expr*, ***kwargs*)

Give the output of *tree* as a multiline string, using line drawings to visualize the hierarchy of expressions (similar to the `tree` unix command line program for showing directory trees)

See also:

qnet.printing.srepr() with `indented=True` produces a similar tree-like rendering of the given expression that can be re-evaluated to the original expression.

qnet.printing.unicodeprinter module

Unicode Printer

Summary

Classes:

QnetUnicodePrinter	Printer for a string (Unicode) representation.
SubSupFmt	A format string that divides into a name, subscript, and superscript
SubSupFmtNoUni	SubSupFmt with default unicode_sub_super=False

Reference

class qnet.printing.unicodeprinter.**SubSupFmt**(*name*, *sub=None*, *sup=None*, *unicode_sub_super=True*)

Bases: object

A format string that divides into a name, subscript, and superscript

```
>>> fmt = SubSupFmt('{name}', sub='({i},{j})', sup='({sup})')
>>> fmt.format(name='alpha', i='mu', j='nu', sup=1)
'α_(μ,ν)^(1)'
>>> fmt = SubSupFmt('{name}', sub='{sub}', sup='({sup})')
>>> fmt.format(name='alpha', sub='1', sup=1)
'α₁¹'
```

format(***kwargs*)

Format and combine the name, subscript, and superscript

class qnet.printing.unicodeprinter.**SubSupFmtNoUni**(*name*, *sub=None*, *sup=None*, *unicode_sub_super=False*)

Bases: *qnet.printing.unicodeprinter.SubSupFmt*

SubSupFmt with default unicode_sub_super=False

class qnet.printing.unicodeprinter.**QnetUnicodePrinter**(*cache=None*, *settings=None*)

Bases: *qnet.printing.asciiprinter.QnetAsciiPrinter*

Printer for a string (Unicode) representation.

sympy_printer_cls

alias of *qnet.printing.sympy.SympyUnicodePrinter*

printmethod = '_unicode'

Summary

`__all__` Functions:

ascii	Return an ASCII representation of the given object / expression
configure_printing	Context manager for temporarily changing the printing system.
dotprint	Return the **'DOT'_** (graph) description of an Expression tree as a string
init_printing	Initialize the printing system.
latex	Return a LaTeX representation of the given object / expression
print_tree	Print a tree representation of the structure of *expr*
srepr	Render the given expression into a string that can be evaluated in an appropriate context to re-instantiate an identical expression.
tex	Alias for latex()
tree	Give the output of *tree* as a multiline string, using line drawings to visualize the hierarchy of expressions (similar to the tree unix command line program for showing directory trees)
unicode	Return a unicode representation of the given object / expression

Reference

qnet.printing.**init_printing**(*, *reset=False*, *init_sympy=True*, ***kwargs*)

Initialize the printing system.

This determines the behavior of the ascii(), unicode(), and latex() functions, as well as the __str__ and __repr__ of any Expression.

The routine may be called in one of two forms. First,

```
init_printing(
    str_format=<str_fmt>, repr_format=<repr_fmt>,
    caching=<use_caching>, **settings)
```

provides a simplified, "manual" setup with the following parameters.

Parameters

- **str_format** (*str*) – Format for __str__ representation of an Expression. One of 'ascii', 'unicode', 'latex', 'srepr', 'indsrepr' ("indented *srepr*"), or 'tree'. The string representation will be affected by the settings for the corresponding print routine, e.g. unicode() for str_format='unicode'
- **repr_format** (*str*) – Like *str_format*, but for __repr__. This is what gets displayed in an interactive (I)Python session.
- **caching** (*bool*) – By default, the printing functions (ascii(), unicode(), latex()) cache their result for any expression and sub-expression. This is both for efficiency and to give the ability to to supply custom strings for subexpression by passing a *cache* parameter to the printing functions. Initializing the printing system with caching=False disables this possibility.
- **settings** – Any setting understood by any of the printing routines.

Second,

```
init_printing(inifile=<path_to_file>)
```

allows for more detailed settings through a config file, see the *notes on using an INI file*.

If *str_format* or *repr_format* are not given, they will be set to 'unicode' if the current terminal is known to support an UTF8 (accordig to sys.stdout.encoding), and 'ascii' otherwise.

Generally, init_printing() should be called only once at the beginning of a script or notebook. If it is called multiple times, any settings accumulate. To avoid this and to reset the printing system to the defaults, you may pass `reset=True`. In a Jupyter notebook, expressions are rendered graphically via LaTeX, using the settings as they affect the `latex()` printer.

The `sympy.init_printing()` routine is called automatically, unless *init_sympy* is given as `False`.

See also:

`configure_printing()` allows to temporarily change the printing system from what was configured in `init_printing()`.

`qnet.printing.`**`configure_printing`**`(**kwargs)`

Context manager for temporarily changing the printing system.

This takes the same parameters as `init_printing()`

Example

```
>>> A = OperatorSymbol('A', hs=1); B = OperatorSymbol('B', hs=1)
>>> with configure_printing(show_hs_label=False):
...     print(ascii(A + B))
A + B
>>> print(ascii(A + B))
A^(1) + B^(1)
```

`qnet.printing.`**`ascii`**`(expr, cache=None, **settings)`

Return an ASCII representation of the given object / expression

Parameters

- **expr** – Expression to print
- **cache** (*dict or None*) – dictionary to use for caching
- **show_hs_label** (*bool or str*) – Whether to a label for the Hilbert space of *expr*. By default (`show_hs_label=True`), the label is shown as a superscript. It can be shown as a subscript with `show_hs_label='subscript'` or suppressed entirely (`show_hs_label=False`)
- **sig_as_ketbra** (*bool*) – Whether to render instances of `LocalSigma` as a ket-bra (default), or as an operator symbol

Examples

```
>>> A = OperatorSymbol('A', hs=1); B = OperatorSymbol('B', hs=1)
>>> ascii(A + B)
'A^(1) + B^(1)'
>>> ascii(A + B, cache={A: 'A', B: 'B'})
'A + B'
>>> ascii(A + B, show_hs_label='subscript')
'A_(1) + B_(1)'
>>> ascii(A + B, show_hs_label=False)
'A + B'
>>> ascii(LocalSigma(0, 1, hs=1))
'|0><1|^(1)'
>>> ascii(LocalSigma(0, 1, hs=1), sig_as_ketbra=False)
'sigma_0,1^(1)'
```

Note that the accepted parameters and their default values may be changed through `init_printing()` or `configure_printing()`

`qnet.printing.unicode(expr, cache=None, **settings)`
Return a unicode representation of the given object / expression

Parameters

- **expr** – Expression to print
- **cache** (*dict or None*) – dictionary to use for caching
- **show_hs_label** (*bool or str*) – Whether to a label for the Hilbert space of *expr*. By default (`show_hs_label=True`), the label is shown as a superscript. It can be shown as a subscript with `show_hs_label='subscript'` or suppressed entirely (`show_hs_label=False`)
- **sig_as_ketbra** (*bool*) – Whether to render instances of *LocalSigma* as a ket-bra (default), or as an operator symbol
- **unicode_sub_super** (*bool*) – Whether to try to use unicode symbols for sub- or superscripts if possible
- **unicode_op_hats** (*bool*) – Whether to draw unicode hats on single-letter operator symbols

Examples

```
>>> A = OperatorSymbol('A', hs=1); B = OperatorSymbol('B', hs=1)
>>> unicode(A + B)
'Â⁽¹⁾ + B̂⁽¹⁾'
>>> unicode(A + B, cache={A: 'A', B: 'B'})
'A + B'
>>> unicode(A + B, show_hs_label='subscript')
'Â₍₁₎ + B̂₍₁₎'
>>> unicode(A + B, show_hs_label=False)
'Â + B̂'
>>> unicode(LocalSigma(0, 1, hs=1))
'|0⟩⟨1|⁽¹⁾'
>>> unicode(LocalSigma(0, 1, hs=1), sig_as_ketbra=False)
'σ̂_0,1^(1)'
>>> unicode(A + B, unicode_sub_super=False)
'Â^(1) + B̂^(1)'
>>> unicode(A + B, unicode_op_hats=False)
'A⁽¹⁾ + B⁽¹⁾'
```

Note that the accepted parameters and their default values may be changed through *init_printing()* or *configure_printing()*

`qnet.printing.latex(expr, cache=None, **settings)`
Return a LaTeX representation of the given object / expression

Parameters

- **expr** – Expression to print
- **cache** (*dict or None*) – dictionary to use for caching
- **show_hs_label** (*bool or str*) – Whether to a label for the Hilbert space of *expr*. By default (`show_hs_label=True`), the label is shown as a superscript. It can be shown as a subscript with `show_hs_label='subscript'` or suppressed entirely (`show_hs_label=False`)

- **tex_op_macro** (*str*) – macro to use for formatting operator symbols. Must accept 'name' as a format key.
- **tex_textop_macro** (*str*) – macro to use for formatting multi-letter operator names.
- **tex_sop_macro** (*str*) – macro to use for formattign super-operator symbols
- **tex_textsop_macro** (*str*) – macro to use for formatting multi-letter super-operator names
- **tex_identity_sym** (*str*) – macro for the identity symbol
- **tex_use_braket** (*bool*) – If True, use macros from the braket package. Note that this will not automatically render in IPython Notebooks, but it is recommended when generating latex for a document.
- **tex_frac_for_spin_labels** (*bool*) – Whether to use 'frac' when printing basis state labels for spin Hilbert spaces

Examples

```
>>> A = OperatorSymbol('A', hs=1); B = OperatorSymbol('B', hs=1)
>>> latex(A + B)
'\\hat{A}^{(1)} + \\hat{B}^{(1)}'
>>> latex(A + B, cache={A: 'A', B: 'B'})
'A + B'
>>> latex(A + B, show_hs_label='subscript')
'\\hat{A}_{(1)} + \\hat{B}_{(1)}'
>>> latex(A + B, show_hs_label=False)
'\\hat{A} + \\hat{B}'
>>> latex(LocalSigma(0, 1, hs=1))
'\\left\\lvert 0 \\middle\\rangle\\!\\middle\\langle 1 \\right\\rvert^{(1)}'
>>> latex(LocalSigma(0, 1, hs=1), sig_as_ketbra=False)
'\\hat{\\sigma}_{0,1}^{(1)}'
>>> latex(A + B, tex_op_macro=r'\Op{{{name}}}')
'\\Op{A}^{(1)} + \\Op{B}^{(1)}'
>>> CNOT = OperatorSymbol('CNOT', hs=1)
>>> latex(CNOT)
'\\text{CNOT}^{(1)}'
>>> latex(CNOT, tex_textop_macro=r'\Op{{{name}}}')
'\\Op{CNOT}^{(1)}'
```

```
>>> A = SuperOperatorSymbol('A', hs=1)
>>> latex(A)
'\\mathrm{A}^{(1)}'
>>> latex(A, tex_sop_macro=r'\SOp{{{name}}}')
'\\SOp{A}^{(1)}'
>>> Lindbladian = SuperOperatorSymbol('Lindbladian', hs=1)
>>> latex(Lindbladian)
'\\mathrm{Lindbladian}^{(1)}'
>>> latex(Lindbladian, tex_textsop_macro=r'\SOp{{{name}}}')
'\\SOp{Lindbladian}^{(1)}'
```

```
>>> latex(IdentityOperator)
'\\mathbb{1}'
>>> latex(IdentityOperator, tex_identity_sym=r'\identity')
'\\identity'
```

(continues on next page)

(continued from previous page)

```
>>> latex(LocalSigma(0, 1, hs=1), tex_use_braket=True)
'\\Ket{0}\\!\\Bra{1}^{(1)}'
```

```
>>> spin = SpinSpace('s', spin=(1, 2))
>>> up = SpinBasisKet(1, 2, hs=spin)
>>> latex(up)
'\\left\\lvert +1/2 \\right\\rangle^{(s)}'
>>> latex(up, tex_frac_for_spin_labels=True)
'\\left\\lvert +\\frac{1}{2} \\right\\rangle^{(s)}'
```

Note that the accepted parameters and their default values may be changed through `init_printing()` or `configure_printing()`

`qnet.printing.tex`(*expr*, *cache=None*, ***settings*)
Alias for `latex()`

`qnet.printing.srepr`(*expr*, *indented=False*, *cache=None*)
Render the given expression into a string that can be evaluated in an appropriate context to re-instantiate an identical expression. If *indented* is False (default), the resulting string is a single line. Otherwise, the result is a multiline string, and each positional and keyword argument of each *Expression* is on a separate line, recursively indented to produce a tree-like output. The *cache* may be used to generate more readable expressions.

Example

```
>>> hs = LocalSpace('1')
>>> A = OperatorSymbol('A', hs=hs); B = OperatorSymbol('B', hs=hs)
>>> expr = A + B
>>> srepr(expr)
"OperatorPlus(OperatorSymbol('A', hs=LocalSpace('1')), OperatorSymbol('B',␣
↪hs=LocalSpace('1')))"
>>> eval(srepr(expr)) == expr
True
>>> srepr(expr, cache={hs:'hs'})
"OperatorPlus(OperatorSymbol('A', hs=hs), OperatorSymbol('B', hs=hs))"
>>> eval(srepr(expr, cache={hs:'hs'})) == expr
True
>>> print(srepr(expr, indented=True))
OperatorPlus(
    OperatorSymbol(
        'A',
        hs=LocalSpace(
            '1')),
    OperatorSymbol(
        'B',
        hs=LocalSpace(
            '1')))
>>> eval(srepr(expr, indented=True)) == expr
True
```

See also:

`print_tree()`, respectively `qnet.printing.tree.tree()`, produces an output similar to the indented `srepr()`, for interactive use. Their result cannot be evaluated and the exact output depends on `init_printing()`.

`dotprint()` provides a way to graphically explore the tree structure of an expression.

9.1.4 qnet.utils package

Auxiliary utilities, mostly for internal use

Submodules:

qnet.utils.check_rules module

Utilities for algebraic rules

Summary

Functions:

check_rules_dict	Verify the *rules* that classes may use for the *_rules* or *_binary_rules* class attribute.

Reference

qnet.utils.check_rules.**check_rules_dict**(*rules*)
Verify the *rules* that classes may use for the *_rules* or *_binary_rules* class attribute.

Specifically, *rules* must be a OrderedDict-compatible object (list of key-value tuples, dict, OrderedDict) that maps a rule name (str) to a rule. Each rule consists of a *Pattern* and a replaceent callable. The Pattern must be set up to match a *ProtoExpr*. That is, the Pattern should be constructed through the *pattern_head()* routine.

Raises

- TypeError – If *rules* is not compatible with OrderedDict, the keys in *rules* are not strings, or rule is not a tuple of (*Pattern*, *callable*)
- ValueError – If the *head*-attribute of each Pattern is not an instance of *ProtoExpr*, or if there are duplicate keys in *rules*

Returns OrderedDict of rules

qnet.utils.containers module

Tools for working with data structures built from native containers.

Summary

Functions:

nested_tuple	Recursively transform a container structure to a nested tuple.
sorted_if_possible	Create a sorted list of elements of an iterable if they are orderable.

Reference

qnet.utils.containers.**sorted_if_possible**(*iterable*, ***kwargs*)
Create a sorted list of elements of an iterable if they are orderable.

See *sorted* for details on optional arguments to customize the sorting.

Parameters

- **iterable** (*Iterable*) – Iterable returning a finite number of elements to sort.
- **kwargs** – Keyword arguments are passed on to *sorted*.

Returns List of elements, sorted if orderable, otherwise kept in the order of iteration.

Return type list

qnet.utils.containers.**nested_tuple**(*container*)
Recursively transform a container structure to a nested tuple.

The function understands container types inheriting from the selected abstract base classes in *collections.abc*, and performs the following replacements: *Mapping*

tuple of key-value pair *tuple's. The order is preserved in the case of an 'OrderedDict*, otherwise the key-value pairs are sorted if orderable and otherwise kept in the order of iteration.

Sequence *tuple* containing the same elements in unchanged order.

***Container and Iterable and Sized* (equivalent to *Collection* in python >= 3.6)** *tuple* containing the same elements in sorted order if orderable and otherwise kept in the order of iteration.

The function recurses into these container types to perform the same replacement, and leaves objects of other types untouched.

The returned container is hashable if and only if all the values contained in the original data structure are hashable.

Parameters **container** – Data structure to transform into a nested tuple.

Returns Nested tuple containing the same data as *container*.

Return type tuple

qnet.utils.indices module

Summary

Classes:

FockIndex	Symbolic index labeling a basis state in a *LocalSpace*
FockLabel	Symbolic label that evaluates to the label of a basis state
IdxSym	Index symbol in an indexed sum or product
IndexOverFockSpace	Index range over the integer indices of a *LocalSpace* basis
IndexOverList	Index over a list of explicit values
IndexOverRange	Index over the inclusive range between two integers
IndexRangeBase	Base class for index ranges

Continued on next page

Table 53 – continued from previous page

IntIndex	A symbolic label that evaluates to an integer
SpinIndex	Symbolic label for a spin degree of freedom
StrLabel	Symbolic label that evaluates to a string
SymbolicLabelBase	Base class for symbolic labels

Functions:

product	Cartesian product akin to `itertools.product()`, but accepting generator functions
yield_from_ranges	

`__all__`: *FockIndex*, *FockLabel*, *IdxSym*, *IndexOverFockSpace*, *IndexOverList*, *IndexOverRange*, *IntIndex*, *SpinIndex*, *StrLabel*

Reference

`qnet.utils.indices.`**`product`**`(`**generators, repeat=1*`)`

Cartesian product akin to `itertools.product()`, but accepting generator functions

Unlike `itertools.product()` this function does not convert the input iterables into tuples. Thus, it can handle large or infinite inputs. As a drawback, however, it only works with "restartable" iterables (something that `iter()` can repeatably turn into an iterator, or a generator function (but not the generator iterator that is returned by that generator function)

Parameters

- **`generators`** – list of restartable iterators or generator functions
- **`repeat`** – number of times *generators* should be repeated

Adapted from https://stackoverflow.com/q/12093364/

`qnet.utils.indices.`**`yield_from_ranges`**`(`*ranges*`)`

`class` `qnet.utils.indices.`**`IdxSym`**

Bases: `sympy.core.symbol.Symbol`

Index symbol in an indexed sum or product

Parameters

- **`name`** (*str*) – The label for the symbol. It must be a simple Latin or Greek letter, possibly with a subscript, e.g. `'i'`, `'mu'`, `'gamma_A'`
- **`primed`** (*int*) – Number of prime marks (‘) associated with the symbol

Notes

The symbol can be used in arbitrary algebraic (sympy) expressions:

```
>>> sympy.sqrt(IdxSym('n') + 1)
sqrt(n + 1)
```

By default, the symbol is assumed to represent an integer. If this is not the case, you can instantiate explicitly as a non-integer:

```
>>> IdxSym('i').is_integer
True
>>> IdxSym('i', integer=False).is_integer
False
```

You may also declare the symbol as positive:

```
>>> IdxSym('i').is_positive
>>> IdxSym('i', positive=True).is_positive
True
```

The *primed* parameter is used to automatically create distinguishable indices in products of sums, or more generally if the same index occurs in an expression with potentially differnt values:

```
>>> ascii(IdxSym('i', primed=2))
"i''"
>>> IdxSym('i') == IdxSym('i', primed=1)
False
```

It should not be used when creating indices "by hand"

Raises

- `ValueError` – if *name* is not a simple symbol label, or if primed < 0
- `TypeError` – if *name* is not a string

is_finite = True

is_Symbol = True

is_symbol = True

is_Atom = True

primed

incr_primed(*incr=1*)
Return a copy of the index with an incremented *primed*

prime
equivalent to `inc_primed()` with `incr=1`

default_assumptions = {'finite': True, 'infinite': False}

is_infinite = False

class qnet.utils.indices.**SymbolicLabelBase**(*expr*)
Bases: `object`

Base class for symbolic labels

A symbolic label is a SymPy expression that contains one or more *IdxSym*, and can be rendered into an integer or string label by substituting integer values for each *IdxSym*.

See *IntIndex* for an example.

substitute(*var_map*)
Substitute in the expression describing the label.

If the result of the substitution no longer contains any *IdxSym*, this returns a "rendered" label.

free_symbols
Free symbols in the expression describing the label

class qnet.utils.indices.**IntIndex**(*expr*)
Bases: *qnet.utils.indices.SymbolicLabelBase*

A symbolic label that evaluates to an integer

The label can be rendered via `substitute()`:

```
>>> i, j = symbols('i, j', cls=IdxSym)
>>> idx = IntIndex(i+j)
>>> idx.substitute({i: 1, j:1})
2
```

An "incomplete" substitution (anything that still leaves a *IdxSym* in the label expression) will result in another *IntIndex* instance:

```
>>> idx.substitute({i: 1})
IntIndex(Add(IdxSym('j', integer=True), Integer(1)))
```

class qnet.utils.indices.**FockIndex**(*expr*)
Bases: *qnet.utils.indices.IntIndex*

Symbolic index labeling a basis state in a *LocalSpace*

fock_index

class qnet.utils.indices.**StrLabel**(*expr*)
Bases: *qnet.utils.indices.SymbolicLabelBase*

Symbolic label that evaluates to a string

Example

```
>>> i = symbols('i', cls=IdxSym)
>>> A = symbols('A', cls=sympy.IndexedBase)
>>> lbl = StrLabel(A[i])
>>> lbl.substitute({i: 1})
'A_1'
```

class qnet.utils.indices.**FockLabel**(*expr*, *hs*)
Bases: *qnet.utils.indices.StrLabel*

Symbolic label that evaluates to the label of a basis state

This evaluates first to an index, and then to the label for the basis state of the Hilbert space for that index:

```
>>> hs = LocalSpace('tls', basis=('g', 'e'))
>>> i = symbols('i', cls=IdxSym)
>>> lbl = FockLabel(i, hs=hs)
>>> lbl.substitute({i: 0})
'g'
```

fock_index

substitute(*var_map*)
Substitute in the expression describing the label.

If the result of the substitution no longer contains any *IdxSym*, this returns a "rendered" label.

class qnet.utils.indices.**SpinIndex**(*expr*, *hs*)
Bases: *qnet.utils.indices.StrLabel*

Symbolic label for a spin degree of freedom

This evaluates to a string representation of an integer or half-integer. For values of e.g. 1, -1, 1/2, -1/2, the rendered resulting string is "+1", "-1", "+1/2", "-1/2", respectively (in agreement with the convention for the basis labels in a spin degree of freedom)

```
>>> i = symbols('i', cls=IdxSym)
>>> hs = SpinSpace('s', spin='1/2')
>>> lbl = SpinIndex(i/2, hs)
>>> lbl.substitute({i: 1})
'+1/2'
```

Rendering an expression that is not integer or half-integer valued results in a `ValueError`.

fock_index

substitute(*var_map*)
Substitute in the expression describing the label.

If the result of the substitution no longer contains any *IdxSym*, this returns a "rendered" label.

class qnet.utils.indices.**IndexRangeBase**(*index_symbol*)
Bases: `object`

Base class for index ranges

Index ranges occur in indexed sums or products.

iter()

substitute(*var_map*)

piecewise_one(*expr*)
Value of 1 for all index values in the range, 0 otherwise

A `Piecewise` object that is 1 for any value of *expr* in the range of possible index values, and 0 otherwise.

class qnet.utils.indices.**IndexOverList**(*index_symbol*, *values*)
Bases: *qnet.utils.indices.IndexRangeBase*

Index over a list of explicit values

Parameters

- **index_symbol** (`IdxSym`) – The symbol iterating over the value
- **values** (*list*) – List of values for the index

iter()

substitute(*var_map*)

piecewise_one(*expr*)
Value of 1 for all index values in the range, 0 otherwise

A `Piecewise` object that is 1 for any value of *expr* in the range of possible index values, and 0 otherwise.

class qnet.utils.indices.**IndexOverRange**(*index_symbol*, *start_from*, *to*, *step=1*)
Bases: *qnet.utils.indices.IndexRangeBase*

Index over the inclusive range between two integers

Parameters

- **index_symbol** (`IdxSym`) – The symbol iterating over the range
- **start_from** (*int*) – Starting value for the index

- **to** (*int*) – End value of the index
- **step** (*int*) – Step width by which index increases

iter()

range

substitute(*var_map*)

piecewise_one(*expr*)
Value of 1 for all index values in the range, 0 otherwise

A Piecewise object that is 1 for any value of *expr* in the range of possible index values, and 0 otherwise.

class qnet.utils.indices.**IndexOverFockSpace**(*index_symbol*, *hs*)
Bases: *qnet.utils.indices.IndexRangeBase*

Index range over the integer indices of a *LocalSpace* basis

Parameters

- **index_symbol** (IdxSym) – The symbol iterating over the range
- **hs** (LocalSpace) – Hilbert space over whose basis to iterate

iter()

substitute(*var_map*)

piecewise_one(*expr*)
Value of 1 for all index values in the range, 0 otherwise

A Piecewise object that is 1 for any value of *expr* in the range of possible index values, and 0 otherwise.

qnet.utils.ordering module

The *ordering* package implements the default canonical ordering for sums and products of operators, states, and superoperators.

To the extent that commutativity rules allow this, the ordering defined here groups objects of the same Hilbert space together, and orders these groups in the same order that the Hilbert spaces occur in a *ProductSpace* (lexicographically/by *order_index*/by complexity). Objects within the same Hilbert space (again, assuming they commute) are ordered by the *KeyTuple* value that *expr_order_key* returns for each object. Note that *expr_order_key* defers to the object's *_order_key* property, if available. This property should be defined for all QNET Expressions, generally ordering objects according to their type, then their label (if any), then their pre-factor then any other properties.

We assume that quantum operations have either full commutativity (sums, or products of states), or commutativity of objects only in different Hilbert spaces (e.g. products of operators). The former is handled by *FullCommutativeHSOrder*, the latter by *DisjunctCommutativeHSOrder*. Theses classes serve as the *order_key* for sums and products (e.g. *OperatorPlus* and similar classes)

A user may implement a custom ordering by subclassing (or replacing) *FullCommutativeHSOrder* and/or *DisjunctCommutativeHSOrder*, and assigning their replacements to all the desired algebraic classes.

Summary

Classes:

DisjunctCommutativeHSOrder	Auxiliary class that generates the correct pseudo-order relation for operator products.
FullCommutativeHSOrder	Auxiliary class that generates the correct pseudo-order relation for operator sums.
KeyTuple	A tuple that allows for ordering, facilitating the default ordering of Operations.

Functions:

expr_order_key	A default order key for arbitrary expressions

Reference

class qnet.utils.ordering.**KeyTuple**

Bases: `tuple`

A tuple that allows for ordering, facilitating the default ordering of Operations. It differs from a normal tuple in that it falls back to string comparison if any elements are not directly comparable

qnet.utils.ordering.**expr_order_key**(*expr*)

A default order key for arbitrary expressions

class qnet.utils.ordering.**DisjunctCommutativeHSOrder**(*op*, *space_order=None*, *op_order=None*)

Bases: `object`

Auxiliary class that generates the correct pseudo-order relation for operator products. Only operators acting on disjoint Hilbert spaces are commuted to reflect the order the local factors have in the total Hilbert space. I.e., `sorted(factors, key=DisjunctCommutativeHSOrder)` achieves this ordering.

class qnet.utils.ordering.**FullCommutativeHSOrder**(*op*, *space_order=None*, *op_order=None*)

Bases: `object`

Auxiliary class that generates the correct pseudo-order relation for operator sums. Operators are first ordered by their Hilbert space, then by their order-key; `sorted(factors, key=FullCommutativeHSOrder)` achieves this ordering.

qnet.utils.permutations module

Summary

Exceptions:

BadPermutationError	Can be raised to signal that a permutation does not pass the :py:func:check_permutation test.

Functions:

block_perm_and_perms_within_blocks	Decompose a permutation into a block permutation and into permutations acting within each block.

Continued on next page

Table 58 – continued from previous page

check_permutation	Verify that a tuple of permutation image points `(sigma(1), sigma(2), ..., sigma(n))` is a valid permutation, i.e.
compose_permutations	Find the composite permutation
concatenate_permutations	Concatenate two permutations:
full_block_perm	Extend a permutation of blocks to a permutation for the internal signals of all blocks.
invert_permutation	Compute the image tuple of the inverse permutation.
permutation_from_block_permutations	Reverse operation to *permutation_to_block_permutations()* Compute the concatenation of permutations
permutation_from_disjoint_cycles	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
permutation_to_block_permutations	If possible, decompose a permutation into a sequence of permutations each acting on individual ranges of the full range of indices.
permutation_to_disjoint_cycles	Any permutation sigma can be represented as a product of cycles.
permute	Apply a permutation sigma({j}) to an arbitrary sequence.

Reference

exception qnet.utils.permutations.**BadPermutationError**

Bases: ValueError

Can be raised to signal that a permutation does not pass the :py:func:check_permutation test.

qnet.utils.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.utils.permutations.**invert_permutation**(*permutation*)

Compute the image tuple of the inverse permutation.

Parameters **permutation** – A valid (cf. :py:func:check_permutation) permutation.

Returns The inverse permutation tuple

Return type tuple

qnet.utils.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.utils.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.utils.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.utils.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.utils.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.utils.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.utils.permutations.`**`permute`**(*sequence*, *permutation*)

Apply a permutation sigma({j}) to an arbitrary sequence.

Parameters

- **sequence** – Any finite length sequence `[l_1,l_2,...l_n]`. If it is a list, tuple or str, the return type will be the same.
- **permutation** (*tuple*) – permutation image tuple

Returns The permuted sequence `[l_sigma(1), l_sigma(2), ..., l_sigma(n)]`

Raise BadPermutationError or ValueError

`qnet.utils.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.utils.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.utils.properties_for_args module

Class decorator for adding properties for arguments

Summary

Functions:

properties_for_args	For a class with an attribute *arg_names* containing a list of names, add a property for every name in that list.

Reference

qnet.utils.properties_for_args.**properties_for_args**(*cls*, *arg_names='_arg_names'*)

For a class with an attribute *arg_names* containing a list of names, add a property for every name in that list.

It is assumed that there is an instance attribute `self._<arg_name>`, which is returned by the *arg_name* property. The decorator also adds a class attribute `_has_properties_for_args` that may be used to ensure that a class is decorated.

qnet.utils.singleton module

Constant algebraic objects are best implemented as singletons (i.e., they only exist as a single object). This module provides the means to declare singletons:

- The *Singleton* metaclass ensures that every class based on it produces the same object every time it is instantiated
- The *singleton_object()* class decorator converts a singleton class definition into the actual singleton object

Singletons in QNET should use both of these.

Note: In order for the Sphinx autodoc extension to correctly recognize singletons, a custom documenter will have to be registered. The Sphinx `conf.py` file must contain the following:

```
from sphinx.ext.autodoc import DataDocumenter

class SingletonDocumenter(DataDocumenter):
    directivetype = 'data'
    objtype = 'singleton'
    priority = 20

    @classmethod
    def can_document_member(cls, member, membername, isattr, parent):
        return isinstance(member, qnet.utils.singleton.SingletonType)

def setup(app):
    # ... (other hook settings)
    app.add_autodocumenter(SingletonDocumenter)
```

Summary

Classes:

Singleton	Metaclass for singletons

Functions:

singleton_object	Class decorator that transforms (and replaces) a class definition (which must have a Singleton metaclass) with the actual singleton object.

Data:

SingletonType	A dummy type that may be used to check whether an object is a Singleton.

`__all__`: *Singleton*, *SingletonType*, *singleton_object*

Reference

qnet.utils.singleton.**singleton_object**(*cls*)

Class decorator that transforms (and replaces) a class definition (which must have a Singleton metaclass) with the actual singleton object. Ensures that the resulting object can still be "instantiated" (i.e., called), returning the same object. Also ensures the object can be pickled, is hashable, and has the correct string representation (the name of the singleton)

If the class defines a *_hash_val* class attribute, the hash of the singleton will be the hash of that value, and the singleton will compare equal to that value. Otherwise, the singleton will have a unique hash and compare equal only to itself.

class qnet.utils.singleton.**Singleton**

Bases: abc.ABCMeta

Metaclass for singletons

Any instantiation of a singleton class yields the exact same object, e.g.:

```
>>> class MyClass(metaclass=Singleton):
...     pass
>>> a = MyClass()
>>> b = MyClass()
>>> a is b
True
```

You can check that an object is a singleton using:

```
>>> isinstance(a, SingletonType)
True
```

qnet.utils.singleton.**SingletonType = <class 'qnet.utils.singleton.SingletonType'>**

A dummy type that may be used to check whether an object is a Singleton:

```
isinstance(obj, SingletonType)
```

qnet.utils.testing module

Collection of routines needed for testing. This includes proto-fixtures, i.e. routines that should be imported and then turned into a fixture with the pytest.fixture decorator.

See <https://pytest.org/latest/fixture.html>

Summary

Classes:

QnetAsciiTestPrinter	A Printer subclass for testing

Functions:

check_idempotent_create	Check that an expression is 'idempotent'
datadir	Proto-fixture responsible for searching a folder with the same name of test module and, if available, moving all contents to a temporary directory so tests can use them freely.

Reference

class qnet.utils.testing.**QnetAsciiTestPrinter**(*cache=None*, *settings=None*)

Bases: *qnet.printing.asciiprinter.QnetAsciiPrinter*

A Printer subclass for testing

qnet.utils.testing.**datadir**(*tmpdir*, *request*)

Proto-fixture responsible for searching a folder with the same name of test module and, if available, moving all contents to a temporary directory so tests can use them freely.

In any test, import the datadir routine and turn it into a fixture:

```
>>> import pytest
>>> import qnet.utils.testing
>>> datadir = pytest.fixture(qnet.utils.testing.datadir)
```

qnet.utils.testing.**check_idempotent_create**(*expr*)

Check that an expression is 'idempotent'

qnet.utils.unicode module

Utils for working with unicode strings

Summary

Functions:

grapheme_len	Number of graphemes in *text*
ljust	Left-justify text to a total of *width*
rjust	Right-justify text for a total of *width* graphemes

Reference

qnet.utils.unicode.**grapheme_len**(*text*)

Number of graphemes in *text*

This is the length of the *text* when printed::

```
>>> s = 'A'
>>> len(s)
2
>>> grapheme_len(s)
1
```

qnet.utils.unicode.**ljust**(*text*, *width*, *fillchar=' '*)

Left-justify text to a total of *width*

The *width* is based on graphemes:

```
>>> s = 'A'
>>> s.ljust(2)
'A'
>>> ljust(s, 2)
'A '
```

qnet.utils.unicode.**rjust**(*text*, *width*, *fillchar=' '*)

Right-justify text for a total of *width* graphemes

The *width* is based on graphemes:

```
>>> s = 'A'
>>> s.rjust(2)
'A'
>>> rjust(s, 2)
' A'
```

Summary

`__all__` Classes:

FockIndex	Symbolic index labeling a basis state in a *LocalSpace*
FockLabel	Symbolic label that evaluates to the label of a basis state
IdxSym	Index symbol in an indexed sum or product
IndexOverFockSpace	Index range over the integer indices of a *LocalSpace* basis
IndexOverList	Index over a list of explicit values
IndexOverRange	Index over the inclusive range between two integers
IntIndex	A symbolic label that evaluates to an integer
Singleton	Metaclass for singletons
SpinIndex	Symbolic label for a spin degree of freedom
StrLabel	Symbolic label that evaluates to a string

`__all__` Functions:

singleton_object	Class decorator that transforms (and replaces) a class definition (which must have a Singleton metaclass) with the actual singleton object.

`__all__` Data:

SingletonType	A dummy type that may be used to check whether an object is a Singleton:

9.1.5 qnet.visualization package

Visualization routines, e.g. circuit diagrams.

Submodules:

qnet.visualization.circuit_pyx module

Circuit visualization via the pyx package

This requires a working LaTeX installation.

Summary

Functions:

draw_circuit	Generate a graphic representation of circuit and store them in a file.
draw_circuit_canvas	Generate a PyX graphical representation of a circuit expression object.

`__all__`: *draw_circuit*, *draw_circuit_canvas*

Reference

qnet.visualization.circuit_pyx.**draw_circuit_canvas**(*circuit*, *hunit=4*, *vunit=-1.0*, *rhmargin=0.1*, *rvmargin=0.2*, *rpermutation_length=0.4*, *draw_boxes=True*, *permutation_arrows=False*)

Generate a PyX graphical representation of a circuit expression object.

Parameters

- **circuit** (*ca.Circuit*) – The circuit expression
- **hunit** (*float*) – The horizontal length unit, default = `HUNIT`
- **vunit** (*float*) – The vertical length unit, default = `VUNIT`
- **rhmargin** (*float*) – relative horizontal margin, default = `RHMARGIN`
- **rvmargin** (*float*) – relative vertical margin, default = `RVMARGIN`

- **rpermutation_length** (*float*) – the relative length of a permutation circuit, default = `RPLENGTH`
- **draw_boxes** (*bool*) – Whether to draw indicator boxes to denote subexpressions (Concatenation, SeriesProduct, etc.), default = `True`
- **permutation_arrows** (*bool*) – Whether to draw arrows within the permutation visualization, default = `False`

Returns A PyX canvas object that can be further manipulated or printed to an output image.

Return type pyx.canvas.canvas

`qnet.visualization.circuit_pyx.`**`draw_circuit`**(*circuit, filename, direction='lr', hunit=4, vunit=-1.0, rhmargin=0.1, rvmargin=0.2, rpermutation_length=0.4, draw_boxes=True, permutation_arrows=False*)

Generate a graphic representation of circuit and store them in a file. The graphics format is determined from the file extension.

Parameters

- **circuit** (*ca.Circuit*) – The circuit expression
- **filename** (*str*) – A filepath to store the output image under. The file name suffix determines the output graphics format
- **direction** – The horizontal direction of laying out series products. One of `'lr'` and `'rl'`. This option overrides a negative value for `hunit`, default = `'lr'`
- **hunit** (*float*) – The horizontal length unit, default = `HUNIT`
- **vunit** (*float*) – The vertical length unit, default = `VUNIT`
- **rhmargin** (*float*) – relative horizontal margin, default = `RHMARGIN`
- **rvmargin** (*float*) – relative vertical margin, default = `RVMARGIN`
- **rpermutation_length** (*float*) – the relative length of a permutation circuit, default = `RPLENGTH`
- **draw_boxes** (*bool*) – Whether to draw indicator boxes to denote subexpressions (Concatenation, SeriesProduct, etc.), default = `True`
- **permutation_arrows** (*bool*) – Whether to draw arrows within the permutation visualization, default = `False`

Returns `True` if printing was successful, `False` if not.

Return type bool

Summary

`__all__` Functions:

draw_circuit	Generate a graphic representation of circuit and store them in a file.
draw_circuit_canvas	Generate a PyX graphical representation of a circuit expression object.

9.1.6 Summary

`__all__` Exceptions:

AlgebraError	Base class for all algebraic errors
AlgebraException	Base class for all algebraic exceptions
BadLiouvillianError	Raised when a Liouvillian is not of standard Lindblad form.
BasisNotSetError	Raised if the basis or a Hilbert space dimension is unavailable
CannotConvertToSLH	Raised when a circuit algebra object cannot be converted to SLH
CannotEliminateAutomatically	Raised when attempted automatic adiabatic elimination fails.
CannotSimplify	Raised when a rule cannot further simplify an expression
CannotSymbolicallyDiagonalize	Matrix cannot be diagonalized analytically.
CannotVisualize	Raised when a circuit cannot be visually represented.
IncompatibleBlockStructures	Raised for invalid block-decomposition
InfiniteSumError	Raised when expanding a sum into an infinite number of terms
NoConjugateMatrix	Raised when entries of Matrix have no defined conjugate
NonSquareMatrix	Raised when a Matrix fails to be square
OverlappingSpaces	Raised when objects fail to be in separate Hilbert spaces.
SpaceTooLargeError	Raised when objects fail to be have overlapping Hilbert spaces.
UnequalSpaces	Raised when objects fail to be in the same Hilbert space.
WrongCDimError	Raised for mismatched channel number in circuit series

`__all__` Classes:

Adjoint	Symbolic Adjoint of an operator
BasisKet	Local basis state, identified by index or label
Beamsplitter	Infinite bandwidth beamsplitter component.
Bra	The associated dual/adjoint state for any ket
BraKet	The symbolic inner product between two states
CPermutation	Channel permuting circuit
Circuit	Base class for the circuit algebra elements
CircuitSymbol	Symbolic circuit element
CoherentDriveCC	Coherent displacement of the input field
CoherentStateKet	Local coherent state, labeled by a complex amplitude
Commutator	Commutator of two operators
Component	Base class for circuit components
Concatenation	Concatenation of circuit elements
Create	Bosonic creation operator
Destroy	Bosonic annihilation operator
Displace	Unitary coherent displacement operator
Eq	Symbolic equation
Expression	Base class for all QNET Expressions
Feedback	Feedback on a single channel of a circuit
FockIndex	Symbolic index labeling a basis state in a LocalSpace
FockLabel	Symbolic label that evaluates to the label of a basis state
HilbertSpace	Base class for Hilbert spaces
IdxSym	Index symbol in an indexed sum or product
IndexOverFockSpace	Index range over the integer indices of a LocalSpace basis
IndexOverList	Index over a list of explicit values
IndexOverRange	Index over the inclusive range between two integers
IndexedSum	Base class for indexed sums

Continued on next page

Table 67 – continued from previous page

IntIndex	A symbolic label that evaluates to an integer
Jminus	Lowering operator on a spin space
Jplus	Raising operator of a spin space
Jz	Spin (angular momentum) operator in z-direction
KetBra	Outer product of two states
KetIndexedSum	Indexed sum over Kets
KetPlus	Sum of states
KetSymbol	Symbolic state
LocalKet	A state on a LocalSpace
LocalOperator	Base class for "known" operators on a LocalSpace
LocalSigma	Level flip operator between two levels of a LocalSpace
LocalSpace	Hilbert space for a single degree of freedom.
MatchDict	Result of a Pattern.match()
Matrix	Matrix of Expressions
NullSpaceProjector	Projection operator onto the nullspace of its operand
Operation	Base class for "operations"
Operator	Base class for all quantum operators.
OperatorDerivative	Symbolic partial derivative of an operator
OperatorIndexedSum	Indexed sum over operators
OperatorPlus	Sum of Operators
OperatorPlusMinusCC	An operator plus or minus its complex conjugate
OperatorSymbol	Symbolic operator
OperatorTimes	Product of operators
OperatorTimesKet	Product of an operator and a state.
OperatorTrace	(Partial) trace of an operator
Pattern	Pattern for matching an expression
Phase	Unitary "phase" operator
PhaseCC	Coherent phase shift cicuit component
ProductSpace	Tensor product of local Hilbert spaces
PseudoInverse	Unevaluated pseudo-inverse X^+ of an operator X
QuantumAdjoint	Base class for adjoints of quantum expressions
QuantumDerivative	Symbolic partial derivative
QuantumExpression	Base class for expressions associated with a Hilbert space
QuantumIndexedSum	Base class for indexed sums
QuantumOperation	Base class for operations on quantum expression
QuantumPlus	General implementation of addition of quantum expressions
QuantumSymbol	Symbolic element of an algebra
QuantumTimes	General implementation of product of quantum expressions
SLH	Element of the SLH algebra
SPost	Linear post-multiplication operator
SPre	Linear pre-multiplication operator
Scalar	Base class for Scalars
ScalarDerivative	Symbolic partial derivative of a scalar
ScalarExpression	Base class for scalars with non-scalar arguments
ScalarIndexedSum	Indexed sum over scalars
ScalarPlus	Sum of scalars
ScalarPower	A scalar raised to a power
ScalarTimes	Product of scalars
ScalarTimesKet	Product of a Scalar coefficient and a ket
ScalarTimesOperator	Product of a Scalar coefficient and an Operator

Continued on next page

Table 67 – continued from previous page

ScalarTimesQuantumExpression	Product of a *Scalar* and a QuantumExpression
ScalarTimesSuperOperator	Product of a *Scalar* coefficient and a SuperOperator
ScalarValue	Wrapper around a numeric or symbolic value
SeriesInverse	Symbolic series product inversion operation
SeriesProduct	The series product circuit operation.
SingleQuantumOperation	Base class for operations on a single quantum expression
Singleton	Metaclass for singletons
SpinIndex	Symbolic label for a spin degree of freedom
SpinOperator	Base class for operators in a spin space
SpinSpace	A Hilbert space for an integer or half-integer spin system
Squeeze	Unitary squeezing operator
State	Base class for states in a Hilbert space
StateDerivative	Symbolic partial derivative of a state
StrLabel	Symbolic label that evaluates to a string
SuperAdjoint	Adjoint of a super-operator
SuperOperator	Base class for super-operators
SuperOperatorDerivative	Symbolic partial derivative of a super-operator
SuperOperatorPlus	A sum of super-operators
SuperOperatorSymbol	Symbolic super-operator
SuperOperatorTimes	Product of super-operators
SuperOperatorTimesOperator	Application of a super-operator to an operator
TensorKet	A tensor product of kets

__all__ Functions:

FB	Wrapper for *Feedback*, defaulting to last channel
KroneckerDelta	Kronecker delta symbol
LocalProjector	A projector onto a specific level of a *LocalSpace*
PauliX	Pauli-type X-operator
PauliY	Pauli-type Y-operator
PauliZ	Pauli-type Z-operator
SLH_to_qutip	Generate and return QuTiP representation matrices for the Hamiltonian and the collapse op
SpinBasisKet	Constructor for a BasisKet for a SpinSpace
Sum	Instantiator for an arbitrary indexed sum.
adjoint	Return the adjoint of an obj.
anti_commutator	If B != None, return the anti-commutator $\{A, B\}$, otherwise return the super-operator $\{$
ascii	Return an ASCII representation of the given object / expression
block_matrix	Generate the operator matrix with quadrants
circuit_identity	Return the circuit identity for n channels
commutator	Commutator of *A* and *B*
configure_printing	Context manager for temporarily changing the printing system.
connect	Connect a list of components according to a list of connections.
convert_to_qutip	Convert a QNET expression to a qutip object
convert_to_sympy_matrix	Convert a QNET expression to an explicit n x n instance of *sympy.Matrix*, where n is th
decompose_space	Simplifies OperatorTrace expressions over tensor-product spaces by turning it into iterated
diagm	Generalizes the diagonal matrix creation capabilities of *numpy.diag* to Matrix objects.
dotprint	Return the **'DOT'_** (graph) description of an Expression tree as a string
draw_circuit	Generate a graphic representation of circuit and store them in a file.
draw_circuit_canvas	Generate a PyX graphical representation of a circuit expression object.
eval_adiabatic_limit	Compute the limiting SLH model for the adiabatic approximation

expand_commutators_leibniz	Recursively expand commutators in *expr* according to the Leibniz rule.
extract_channel	Create a CPermutation that extracts channel *k*
factor_coeff	Factor out coefficients of all factors.
factor_for_trace	Given a LocalSpace *ls* to take the partial trace over and an operator *op*, factor the trace
getABCD	Calculate the ABCD-linearization of an SLH model
get_coeffs	Create a dictionary with all Operator terms of the expression (understood as a sum) as keys
hstackm	Generalizes *numpy.hstack* to Matrix objects.
identity_matrix	Generate the N-dimensional identity matrix.
init_algebra	Initialize the algebra system
init_printing	Initialize the printing system.
latex	Return a LaTeX representation of the given object / expression
lindblad	Return the super-operator Lindblad term of the Lindblad operator *C*
liouvillian	Return the Liouvillian super-operator associated with *H* and *Ls*
liouvillian_normal_form	Return a Hamilton operator H and a minimal list of collapse operators Ls that generate the
map_channels	Create a CPermuation based on a dict of channel mappings
match_pattern	Recursively match *expr* with the given *expr_or_pattern*
move_drive_to_H	Move coherent drives from the Lindblad operators to the Hamiltonian.
no_instance_caching	Temporarily disable instance caching in create()
pad_with_identity	Pad a circuit by adding a *n*-channel identity circuit at index *k*
pattern	'Flat' constructor for the Pattern class
pattern_head	Constructor for a Pattern matching a ProtoExpr
prepare_adiabatic_limit	Prepare the adiabatic elimination on an SLH object
print_tree	Print a tree representation of the structure of *expr*
rewrite_with_operator_pm_cc	Try to rewrite expr using OperatorPlusMinusCC
singleton_object	Class decorator that transforms (and replaces) a class definition (which must have a Single
sqrt	Square root of a Scalar or scalar value
srepr	Render the given expression into a string that can be evaluated in an appropriate context to
substitute	Substitute symbols or (sub-)expressions with the given replacements and re-evalute the res
symbols	The symbols() function from SymPy
temporary_instance_cache	Use a temporary cache for instances in create()
temporary_rules	Allow temporary modification of rules for create()
tex	Alias for latex()
tree	Give the output of *tree* as a multiline string, using line drawings to visualize the hierarchy
try_adiabatic_elimination	Attempt to automatically do adiabatic elimination on an SLH object
unicode	Return a unicode representation of the given object / expression
vstackm	Generalizes *numpy.vstack* to Matrix objects.
wc	Constructor for a wildcard-Pattern
zerosm	Generalizes numpy.zeros to Matrix objects.

__all__ Data:

CIdentity	Single pass-through channel; neutral element of `SeriesProduct`
CircuitZero	Zero circuit, the neutral element of `Concatenation`
FullSpace	The 'full space', i.e.
II	`IdentityOperator` constant (singleton) object.
IdentityOperator	`IdentityOperator` constant (singleton) object.
IdentitySuperOperator	Neutral element for product of super-operators
One	The neutral element with respect to scalar multiplication
SingletonType	A dummy type that may be used to check whether an object is a Singleton:
TrivialKet	TrivialKet constant (singleton) object.
TrivialSpace	The 'nullspace', i.e.
Zero	The neutral element with respect to scalar addition
ZeroKet	ZeroKet constant (singleton) object for the null-state.
ZeroOperator	`ZeroOperator` constant (singleton) object.
ZeroSuperOperator	Neutral element for sum of super-operators
tr	Instantiate while applying automatic simplifications

Python Module Index

q

Index

Symbols

A

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

Y

Z