The phfqit package[1]

Philippe Faist *philippe.faist@bluewin.ch*

October 08, 2021

---

[1] *This document corresponds to* phfqit *v4.1, dated 2021/10/08. It is part of the* phfqitltx *package suite, see https://github.com/phfaist/phfqitltx.*

---

phfqit—Utilities to typeset stuff in Quantum Information Theory, in particular general mathematical symbols, operators, and shorthands for entropy measures.

---

1 Introduction

This package provides some useful definitions, mainly for notation of mathematical expressions which are used in quantum information theory (at least by me).

Are included utilities for:

- General symbols and mathematical expressions (identity operator, trace, rank, diagonal, . . .) (section 3)
- Formatting of bits and bit strings (subsection 3.3)
- Formatting of names of logical gates (subsection 3.4)
- Typesetting the names of Lie groups and algebras, for example $\mathfrak{su}(N)$ (section 4)
- Bra-ket notation, and delimited expressions such as average, norm, . . . (subsection 5.2)
- Double bra-ket notation for operators (subsection 5.2)
- Typesetting entropy measures, including the Shannon/von Neumann entropy, the smooth entropies, relative entropies, as well as my coherent relative entropy

■ 2 Basic Usage

This package is straightforward to use:

```
\usepackage{phfqit}
```

A single package option controls which entropy measures are defined for you.

`qitobjdef=⟨stdset | none⟩`

Load the predefined set of "qit objects, " i.e., entropy measures. The entropy measures documented below (and specified as such) will be loaded unless you set `qitobjdef=none`.

`newReIm=⟨true | false⟩`

Do not override LaTeX's default $\Re$ and $\Im$ symbols by Re and Im. See subsection 3.1.

`llanglefrommnsymbolfonts=⟨true | false⟩`

In order to define the operator-kets and operator-bra symbols, we need to have double-angle bracket symbol delimiters loaded as `\llangle` and `\rrangle`. Unlike `\langle` and `\rangle`, they are not provided by default by latex, amsmath or amssymb. What we do is that we load the `\llangle` and `\rrangle` symbols from the MnSymbol fonts in order to define `\oket` and friends. If you would like to provide your own definitions for `\llangle` and `\rrangle`, or if you have problems loading the MnSymbol fonts and don't need the operator-ket symbols, then you can specify `llanglefrommnsymbolfonts=false` and we won't bother loading the MnSymbol fonts. (Note that if you provide your own definitions for `\llangle` and `\rrangle`, they have to be valid delimiters, such that for example the syntax `\left\llangle` is valid.)

Changed in v2.0 [2017/08/16]: Added the `qitobjdef` package option.

Changed in v2.0 [2017/08/16]: Added the `newReIm` package option.

2.1 Semantic vs. Syntactic Notation

The macros in this package are meant to represent a *mathematical quantity*, independently of its final *notation*. For example, `\Hmaxf` indicates corresponds to the "new-style" max-entropy defined with the fidelity,[1] independently of the notation. Then, if the default notation "$H_{\max}$" doesn't suit your taste, you may then simply redefine this command to display whatever you like (see for example instructions in subsection 6.1). This allows to keep better distinction between

---

[1] see Marco Tomamichel, Ph. D., ETH Zurich (2012) arXiv:1203.2142

different measures which may share the same notation in different works of literature. It also allows to switch notation easily, even in documents which use several quantities whose notation may be potentially conflicting.

2.2 Size Specification

Many of the macros in this package allow their delimiters to be sized according to your taste. For example, if there is a large symbol in an entropy measure, say

$$H_{\min}(\bigotimes_i A_i \,|\, B) \ , \tag{1}$$

then it may be necessary to tune the size of the parenthesis delimiters.

This is done with the optional size specification ⟨*size-spec*⟩. The ⟨*size-spec*⟩, whenever it is accepted, is always optional.

The ⟨*size-spec*⟩ starts with the backtick character "`` ` ``", and is followed by a single token which may be a star `*` or a size modifier macro such as `\big`, `\Big`, `\bigg` and `\Bigg`. If the star is specified, then the delimiters are sized with `\left` and `\right`; otherwise the corresponding size modifier is used. When no size specification is present, then the normal character size is used.

For example:

`\Hmin{\bigotimes_i A_i}[B]`	gives	$H_{\min}(\bigotimes_i A_i \,|\, B)$,
``\Hmin`\Big{\bigotimes_i A_i}[B]``	gives	$H_{\min}\Big(\bigotimes_i A_i \,\Big|\, B\Big)$, and
``\Hmin`*{\bigotimes_i A_i}[B]``	gives	$H_{\min}\left(\bigotimes_i A_i \,\middle|\, B\right)$.

■ 3 General Symbols and Math Operators

`\Hs` Hilbert space = $\mathscr{H}$.

`\Ident` Identity operator = $\mathbb{1}$.

`\IdentProc` Identity process. Possible usage syntax is:

`\IdentProc[A][A']{\rho}`	$\mathrm{id}_{A\to A'}(\rho)$
`\IdentProc[A]{\rho}`	$\mathrm{id}_{A}(\rho)$
`\IdentProc[A][A']{}`	$\mathrm{id}_{A\to A'}$
`\IdentProc[A]{}`	id_{A}
`\IdentProc{}`	id
`\IdentProc{\rho}`	$\mathrm{id}(\rho)$
``\IdentProc`\big[A]{\rho}``	$\mathrm{id}_{A}\big(\rho\big)$

This macro accepts a size specification with the backtick (‘`` ` ``’), see subsection 2.2.

`\ee^X` A macro for the exponential. Type the LaTeX code as if `\ee` were just the symbol, i.e. as `\ee^{<ARGUMENT>}`. The ideas is that this macro may be redefined to change the appearance of the e symbol, or even to change the notation to `\exp{<ARGUMENT>}` if needed for inline math.

`\phfqitExpPowerExpression` To change the appearance of whatever you typed as `\ee^{XYZ}`, you can redefine `\phfqitExpPowerExpression`; for instance, to use an upright “e” symbol, you could type:

```
\renewcommand\phfqitExpPowerExpression[1]{\mathrm{e}^{#1}}
```

3.1 Math/Linear Algebra Operators

`\tr`
`\supp`
`\rank`
`\linspan`
`\spec`
`\diag`

Provide some common math operators. The trace tr, the support supp, the rank rank, the linear span span, the spectrum spec and the diagonal matrix diag. (Note that `\span` is already defined by LaTeX, so that we resort to `\linspan`.)

`\Re`
`\Im`

Also, redefine `\Re` and `\Im` (real and imaginary parts of a complex number), to the more readable $\operatorname{Re}(z)$ and $\operatorname{Im}(z)$. (The original symbols were $\Re(z)$ and $\Im(z)$.) Keep the old definitions using the package option `newReIm=false`.

3.2 Poly symbol

`\poly` Can be typeset in $\operatorname{poly}(n)$ time.

3.3 Bits and Bit Strings

`\bit` Format a bit value, for example `\bit{0}` or `\bit0` gives `0` or `1`. This command works both in math mode and text mode.

`\bitstring` Format a bit string. For example `\bitstring{01100101}` is rendered as $\overline{\underline{\mathtt{0\,1\,1\,0\,0\,1\,0\,1}}}$. This command works both in math mode and text mode.

3.4 Logical Gates

`\gate` Format a logical gate. Essentially, this command typesets its argument in small-caps font. For example, with `\gate{C-not}` you get C-NOT. (The default formatting ignores the given capitalization, but if you redefine this command you could exploit this, e.g. by making the “C” in “Cnot” larger than the “not”.)

This command works both in math mode and in text mode.

`\AND`
`\XOR`
`\CNOT`
`\NOT`
`\NOOP`

Some standard gates. These typeset respectively as AND, XOR, C-NOT, NOT, and NO-OP.

4 Lie Groups and Algebras

`\uu(N)`
`\UU(N)`
`\su(N)`
`\SU(N)`
`\so(N)`
`\SO(N)`
`\slalg(N)`
`\SL(N)`
`\GL(N)`
`\SN(N)`
`\phfqitLieGroup`
`\phfqitLieAlgebra`
`\phfqitDiscreteGroup`

Format some common Lie groups and algebras. Note the use of `\slalg` instead of `\sl` because of name conflict with TEX's font command *producing slanted text*.

`\SN(N)` is the symmetric group of N items, and formats by default as S_N.

The macros `\phfqitLieGroup`, `\phfqitLieAlgebra`, and `\phfqitDiscreteGroup` format the name for a standard Lie group, Lie algebra or discrete group along with its argument. Redefine these macros with `\renewcommand` to change the formatting font for Lie groups and algebras for instance. For instance, to format standard Lie groups/algebras and the permutation group with simple italic letters, you can use:

```
\renewcommand\phfqitLieAlgebra[2]{\mathit{#1}({#2})}
\renewcommand\phfqitLieGroup[2]{\mathit{#1}({#2})}
\renewcommand\phfqitDiscreteGroup[2]{\mathit{#1}_{#2}}
```

Changed in v2.0 [2018/02/28]: Added the macro `\GL(N)`.

Changed in v3.0 [2020/07/31]: Added the macros `\slalg(n)`, `\SL(N)`, as well as `\phfqitLieAlgebra`, `\phfqitLieGroup`, and `\phfqitDiscreteGroup`.

5 Bra-Ket Notation and Delimited Expressions

5.1 Bras and kets

All commands here work in math mode only. They all accept a size modifier as described in subsection 2.2. (The size may also be provided as an optional argument; the starred form of the command may also be used to enclose the delimiters with `\left...\right` and have the size determined automatically.) Example usage is:

`\ket{\psi}`	$\lvert\psi\rangle$
`` \ket`\big{\psi} ``, `\ket[\big]{\psi}`	$\big\lvert\psi\big\rangle$
`` \ket`\Big{\psi} ``, `\ket[\Big]{\psi}`	$\Big\lvert\psi\Big\rangle$
`` \ket`\bigg{\psi} ``, `\ket[\bigg]{\psi}`	$\bigg\lvert\psi\bigg\rangle$
`` \ket`\Bigg{\psi} ``, `\ket[\Bigg]{\psi}`	$\Bigg\lvert\psi\Bigg\rangle$
`` \ket`*{\displaystyle\sum_k \psi_k} ``, `\ket*{\displaystyle\sum_k \psi_k}`	$\left\lvert\displaystyle\sum_k \psi_k\right\rangle$

`\ket`	Typeset a quantum mechanical ket. `\ket{\psi}` gives $\lvert\psi\rangle$.
`\bra`	Typeset a bra. `\bra{\psi}` gives $\langle\psi\rvert$.
`\braket`	Typeset a bra-ket inner product. `\braket{\phi}{\psi}` gives $\langle\phi\vert\psi\rangle$.
`\ketbra`	Typeset a ket-bra outer product. `\ketbra{\phi}{\psi}` gives $\lvert\phi\rangle\langle\psi\rvert$.
`\proj`	Typeset a rank-1 projector determined by a ket. `\proj{\psi}` gives $\lvert\psi\rangle\langle\psi\rvert$.
`\matrixel`	Typeset a matrix element. `\matrixel{\phi}{A}{\psi}` gives $\langle\phi\vert A\vert\psi\rangle$.
`\dmatrixel`	Typeset a diagonal matrix element of an operator. `\dmatrixel{\phi}{A}` gives $\langle\phi\vert A\vert\phi\rangle$.
`\innerprod`	Typeset an inner product using the mathematicians' notation. `\innerprod{\phi}{\psi}` gives $\langle\phi,\psi\rangle$.

`\oket`
`\obra`
`\obraket`
`\oketbra`
`\oproj`
`\omatrixel`
`\odmatrixel`
	This package also provides associated double-bra-ket commands as is occasionally used to write "vectors" in Hilbert-Schmidt operator space, such as $\lvert A\rrangle$, $\llangle A\vert\mathbb{1}\rrangle$, $\lvert\mathbb{1}\rrangle\llangle E_k\rvert$, etc. The commands are named `\oket`, `\obra`, `\obraket`, `\oketbra`, `\oproj`, `\omatrixel`, and `\odmatrixel`.
	The commands `\oket`, `\obra`, `\obraket`, etc. offer the same syntax as their corresponding `\ket`, `\bra`, etc. counterparts. For instance, you can type `` \obra`\Big{\sum A_i} `` to obtain $\Big\llangle\sum A_i\Big\rvert$.

`\llangle` `\rrangle`	The phfqit package defines the `\llangle` and `\rrangle` delimiters, by taking the relevant symbols from the MnSymbol fonts. These double-angle bracket symbols are used for the double-bra-ket type constructs (`\oket` and friends). If you'd like to provide your own definition of `\llangle` and `\rrangle`, or if for any reason you would not want us to attempt to load the MnSymbol fonts at all, then you can set the package option `llanglefrommnsymbolfonts=false`. If you provide your own definition of `\llangle` and `\rrangle`, then make sure that they are proper TeX delimiters, i.e., constructs of the form `\left\llangle` … `\right\rrangle` or `\bigl\llangle ... \bigr\rrangle` must work.

If your document never uses the double-bra-ket macros (i.e., none of the `\oket`, `\obra`, and friends), then you may safely specifiy `llanglefrommnsymbolfonts=false` to avoid loading the relevant symbols.

5.2 Delimited expressions: norms, absolute value, etc.

There are also some commonly used delimited expressions defined for convenience.

`\abs` The absolute value of an expression. `\abs{A}` gives $|A|$.

`\avg` The average of an expression. ``\avg`\big{\sum_k A_k}`` gives $\big\langle\sum_k A_k\big\rangle$.

`\norm` The norm of an expression. `\norm{A_k}` gives $\|A_k\|$. (If you'd like to define customized norms, e.g., to add subscripts, then check out the `\phfqitDefineNorm` command discussed below.)

`\intervalc` A closed interval. `\intervalc{x}{y}` gives $[x, y]$.

`\intervalo` An open interval. `\intervalo{x}{y}` gives $]x, y[$.

`\intervalco` A semi-open interval, closed on the lower bound and open on the upper bound. `\intervalco{x}{y}` gives $[x, y[$.

`\intervaloc` A semi-open interval, open on the lower bound and closed on the upper bound. `\intervaloc{x}{y}` gives $]x, y]$.

`\phfqitDefineNorm` The handy command `\phfqitDefineNorm` can be used to define custom norms (e.g. 1-norm, infinity-norm, p/q-norms, etc.). The syntax is `\phfqitDefineNorm`⟨*command name*⟩⟨*before*⟩⟨*after*⟩, for example:

```
\phfqitDefineNorm\onenorm{}{_1}
\phfqitDefineNorm\opnorm{}{_\infty}
```

`\phfqitDeclarePairedDelimiterXWithAltSizing`
`\phfqitDeclarePairedDelimiterXPPWithAltSizing`
For defining more advanced custom delimited expressions, you can use the `\phfqitDeclarePairedDelimiterXWithAltSizing` and `\phfqitDeclarePairedDelimiterXPPWithAltSizing` helpers. These macros wrap the mathtools package's `\DeclarePairedDelimiterX` and `\DeclarePairedDelimiterX` macros, by furthermore enabling the newly defined command to accept the size argument using the backtick syntax described in subsection 2.2. These helpers are used internally to define the commands for kets, bras, norms, etc.

■ 6 Entropy Measures and Other "Qit Objects"

A "Qit Object" is any form of quantity which has several parameters and/or arguments which are put together in some notation. The idea is to use LaTeX macros to represent an actual quantity and not just some set of notational symbols. For example, for the “old” max-entropy $H_{\mathrm{max,old}}(X)_\rho = \log\operatorname{rank}\rho$, you should use `\Hzero` independently of whether it should be denoted by H_0, H_{max} or $H_{\mathrm{max,old}}$. This allows you to change the notation by redefining the command `\Hzero`, while making sure that the correct quantity is addressed. (You might have both “old”-style and “new”-style max-entropy in the same paper. Their meaning should never change, even if you change your mind on the notation.) The macros `\Hmin`, `\Hzero`, `\Hmaxf` and `\HH` may be redefined to change the subscript by using the following code (change “`\mathrm{max},0`” to your favorite subscript text):

```
\renewcommand{\Hzero}{\Hbase{\HSym}{\mathrm{max},0}}
```

The phfqit package provides a basic infrastructure allowing to define such “Qit Object” implementations. This package provides the following Qit Objects: entropy measures (`\Hbase`), an entropy function (`\Hfnbase`), relative entropy measures (`\Dbase`), as well as coherent relative entropy measures (`\DCohbase`). The more specific commands `\Hmin`, `\Hzero`, etc. are then defined based on these “base commands.”

You may also define your own Qit Object implementations. See subsection 6.5 for documentation on that.

The actual entropy measure definitions `\Hmin`, `\Hmaxf`, etc., can be disabled by specifying the package option `qitobjdef=none`.

6.1 Entropy, Conditional Entropy

These entropy measures all share the same syntax. This syntax is only described for the min-entropy `\Hmin`, but the other entropy measures enjoy the same features.

These commands are robust, meaning they can be used for example in figure captions and section headings.

`\Hmin` Min-entropy. The general syntax is `\Hmin`⟨*size-spec*⟩`[`⟨*state*⟩`]` `[`⟨*epsilon*⟩`]` `{`⟨*target system*⟩`}` `[`⟨*conditioning system*⟩`]`. For example:

`\Hmin{X}`	$H_{\mathrm{min}}(X)$
`\Hmin[\rho]{X}`	$H_{\mathrm{min}}(X)_\rho$
`\Hmin[\rho][\epsilon]{X}[Y]`	$H_{\mathrm{min}}^\epsilon(X\,\vert\,Y)_\rho$
`\Hmin[\rho|\rho][\epsilon]{X}[Y]`	$H_{\mathrm{min}}^\epsilon(X\,\vert\,Y)_{\rho\vert\rho}$
`\Hmin[][\epsilon]{X}[Y]`	$H_{\mathrm{min}}^\epsilon(X\,\vert\,Y)$
``\Hmin`\Big[\rho]{X}[Y]``	$H_{\mathrm{min}}^\epsilon\Big(X\,\Big\vert\,Y\Big)_\rho$
``\Hmin`*[\rho]{\bigoplus_i X_i}[Y]``	$H_{\mathrm{min}}^\epsilon\left(\bigoplus_i X_i\,\middle\vert\,Y\right)_\rho$

`\HH` Shannon/von Neumann entropy. This macro has the same arguments as for `\Hmin` (even though, of course, there is no real use in smoothing the Shannon/von Neumann entropy…). For example, `\HH[\rho]{X}[Y]` gives $H(X\,|\,Y)_{\rho}$.

`\Hzero` Rényi-zero max-entropy. This macro has the same arguments as for `\Hmin`. For example, `\Hzero[][\epsilon]{X}[Y]` gives $H_{\mathrm{max},0}^{\epsilon}(X\,|\,Y)$.

`\Hmaxf` The max-entropy. This macro has the same arguments as for `\Hmin`. For example, `\Hmaxf[][\epsilon]{X}[Y]` gives $H_{\mathrm{max}}^{\epsilon}(X\,|\,Y)$.

The commands `\Hmin`, `\HH`, `\Hzero`, and `\Hmaxf` are defined only if the package option `qitobjdef=stdset` is set (which is the default).

`\HSym` You may redefine this macro if you want to change the "H" symbol of all entropy measures. For example, with `\renewcommand\HSym{\spadesuit}`, `\Hmin{A}[B]` would give $\spadesuit_{\mathrm{min}}(A\,|\,B)$.

Appearance and alternative notation.

You may change the notation of any of the above entropy measures by redefining the corresponding commands as follows:

```
\renewcommand{\Hzero}{\Hbase{\HSym}{\mathrm{max}}}
```

Then, `\Hzero[\rho]{A}[B]` would produce: $H_{\mathrm{max}}(A\,|\,B)_{\rho}$.

Base entropy measure macro.

`\Hbase` Base macro entropy for an entropy measure. The general syntax is: `\Hbase{`⟨*H-symbol*⟩`}{`⟨*subscript*⟩`}[`⟨*state*⟩`][`⟨*epsilon*⟩`]{`⟨*target system*⟩`}[`⟨*conditioning system*⟩`]`

Using this macro it is easy to define custom special-purpose entropy measures, for instance:

```
\newcommand\Hxyz{\Hbase{\tilde\HSym}{\mathrm{xyz}}}
```

The above code defines the command `\Hxyz[\rho][\epsilon]{A}[B]` → $\boxed{\tilde{H}_{\mathrm{xyz}}^{\epsilon}(A\,|\,B)_{\rho}}$.

See also the implementation documentation below for more specific information on how to customize parts of the rendering, for instance.

6.2 Entropy Function

`\Hfn` The entropy, written as a mathematical function. It is useful to write, e.g.,

$H(p_1\rho_1 + p_2\rho_2)$ as `\Hfn(p_1\rho_1 + p_2\rho_2)`. Sizing specifications also work, e.g. ``\Hfn`\big(x)`` or ``\Hfn`*(x)``.

Usage is: `\Hfn`⟨*size-spec*⟩`(`⟨*argument*⟩`)`

This macro doesn't allow for any subscript, any epsilon-like superscript nor for any conditioning system. Define your own macro on top of `\Hfnbase` if you need that.

Note that the ⟨*argument*⟩ may contain matching parentheses, e.g., ``\Hfn`\Big( g(x) + h(y) )`` $\rightarrow \boxed{H\Big(g(x)+h(y)\Big)}$.

`\Hfunc` The alias `\Hfunc` is provided for backwards compatibility; same as `\Hfn`.

The commands `\Hfn` and `\Hfunc` are defined only if the package option `qitobjdef=stdset` is set (which is the default).

`\Hfnbase` There is also a base macro for this kind of Qit Object, `\Hfnbase`. It allows you to specify an arbitrary symbol to use for "H", as well as custom subscripts and superscripts. The syntax is:

`\Hfnbase{`⟨*H-symbol*⟩`}{`⟨*sub*⟩`}{`⟨*sup*⟩`}`⟨*size-spec*⟩`(`⟨*argument*⟩`)`.

6.3 Relative Entropy

Relative entropies also have a corresponding set of commands.

The syntax varies from command to command, but all relative entropies accept the final arguments ⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to-state*⟩`}`. The size-spec is as always given using the backtick syntax described in subsection 2.2.

`\DD` Generic relative entropy. The syntax of this command is either of the following:

`\DD`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`,
`\DD_{`⟨*subscript*⟩`}`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`,
`\DD_{`⟨*subscript*⟩`}^{`⟨*superscript*⟩`}`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`,
`\DD^{`⟨*superscript*⟩`}`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`.

In all cases, the argument is typeset as: (⟨*state*⟩ ‖ ⟨*relative-to state*⟩). The size of the delimiters can be set with a size specification using the standard backtick syntax as described in subsection 2.2 (as for the other entropy measures).

Examples:

`\DD{\rho}{\sigma}`	$D(\rho \,\Vert\, \sigma)$
``\DD`*{M_1^\dagger M_1}{\sigma}``	$D\left(M_1^\dagger M_1 \,\middle\Vert\, \sigma\right)$
``\DD`\Big{\rho}{\sigma}``	$D\Big(\rho \,\Big\Vert\, \sigma\Big)$

You can also play around with subscripts and superscripts, but it is recommended to use the macros `\Dminf`, `\Dminz` and `\Dmax` directly. Specifying the subscripts and superscripts to `\DD` should only be done within new custom macros to define new relative entropy measures.

`\DD_{\mathrm{Rob}}^{\epsilon}{\rho}{\sigma}`	$D^{\epsilon}_{\mathrm{Rob}}(\rho\,\|\,\sigma)$
`\DD^{sup}{\rho}{\sigma}`	$D^{sup}(\rho\,\|\,\sigma)$

`\Dmax` The max-relative entropy. The syntax is `\Dmax[`⟨*epsilon*⟩`]`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`

For example `\Dmax[\epsilon]{\rho}{\sigma}` gives $D^{\epsilon}_{\mathrm{max}}(\rho\,\|\,\sigma)$ and ``\Dmax[\epsilon]`\big{\rho}{\sigma}`` gives $D^{\epsilon}_{\mathrm{max}}\big(\rho\,\big\|\,\sigma\big)$.

`\Dminz` The “old” min-relative entropy, based on the Rényi-zero relative entropy. The syntax is the same as for `\Dmax`.

`\Dminf` The “new” min-relative entropy, defined using the fidelity. The syntax is the same as for `\Dmax`.

`\Dr` The Rob-relative entropy. The syntax is the same as for `\Dmax`.

`\DHyp`
`\Dhyp` The hypothesis testing relative entropy (two possible variants). The syntax is the same as for `\Dmax`, except that by default the optional argument is `\eta`. The symbols ‘H’ and ‘h’ are used as subscripts, respectively, for `\DHyp` and `\Dhyp`. Examples: The code `\DHyp{\rho}{\sigma}` gives $D^{\eta}_{\mathrm{H}}(\rho\,\|\,\sigma)$ and `\Dhyp[\eta']{\rho}{\sigma}` gives $D^{\eta'}_{\mathrm{h}}(\rho\,\|\,\sigma)$. (This is because this quantity is directly defined with a η (or ϵ) built in, and it is not a zero-error quantity which is smoothed with the purified distance.)

The commands `\DD`, `\Dmax`, `\Dminz`, `\Dminf`, `\Dr`, `\DHyp` and `\Dhyp` are defined only if the package option `qitobjdef=stdset` is set (which is the default).

Changed in v3.1 [2021/07/27]: Added the `\Dhyp` variant of the hypothesis testing relative entropy.

`\DSym` The symbol to use to denote a relative entropy. You may redefine this command to change the symbol. (This works like `\HSym` above.)

Appearance and alternative notation

You may change the notation of any of the above relative entropy measures by redefining the corresponding commands as follows:

```
\renewcommand{\Dminz}[1][]{\Dbase{\DSym}_{\mathrm{MIN}}^{#1}}
```

The above command produces: `\Dminz[\epsilon]{\rho}{\sigma}` → $\boxed{D^{\epsilon}_{\mathrm{MIN}}(\rho\,\|\,\sigma)}$.

Base relative entropy command

As for the H-type entropy measures, there is a “base relative entropy command” `\Dbase`. Its syntax is:

`\Dbase{`⟨*D-symbol*⟩`}[_{`⟨*subscript*⟩`}][^{`⟨*superscript*⟩`}]`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative-to state*⟩`}`

Example: `\Dbase{\hat\DSym}_{0}^{\eta’}‘\Big{\rho}{\sigma}` → $\boxed{\hat{D}_0^{\eta'}\Big(\rho\,\Big\|\,\sigma\Big)}$

The “`_{`⟨*subscript*⟩`}`” and “`^{`⟨*superscript*⟩`}`” parts are optional and may be specified in any order.

See also the implementation documentation below for more specific information on how to customize parts of the rendering, for instance.

6.4 Coherent Relative Entropy

A macro for the coherent relative entropy is also available.

`\DCohx` Typeset a coherent relative entropy using an alternative form for the reference system. The syntax is:

`\DCohx[`⟨*epsilon*⟩`]`⟨*size-spec*⟩`{`⟨*rho*⟩`}{`⟨*X*⟩`}{`⟨*X’*⟩`}{`⟨Γ_X⟩`}{`⟨$\Gamma_{X'}$⟩`}`

For example, `\DCohx[\epsilon]{\rho}{X}{X’}{\Gamma_X}{\Gamma_{X’}}` gives $\bar{D}^{\epsilon}_{X\to X'}(\rho_{X'R_X}\,\|\,\Gamma_X,\Gamma_{X'})$.

The subscript $X'R_X$ (or whatever the system names) is automatically added to the ⟨*rho*⟩ argument. The ‘R’ symbol is used by default for designating the reference system; you may change that by redefining `\DCohxRefSystemName` (see below). If no subscript should be added to the ⟨*rho*⟩ argument, then begin the ⟨*rho*⟩ argument with a star. For example, `\DCoh{*\sigma_R\otimes\rho_{X’}}{X}{X’}{\Gamma_X}{\Gamma_{X’}}` gives $\bar{D}_{X\to X'}(\sigma_R\otimes\rho_{X'}\,\|\,\Gamma_X,\Gamma_{X'})$.

The ⟨*size-spec*⟩ is of course optional and follows the same syntax as everywhere else (subsection 2.2).

The command `\DCohx` is defined only if the package option `qitobjdef=stdset` is set (which is the default).

`\emptysystem` Use the `\emptysystem` macro to denote a trivial system. For example, `\DCoh{\rho}{X}{\emptysystem}{\Gamma}{1}` gives $\bar{D}_{X\to\emptyset}(\rho_X\,\|\,\Gamma,1)$.

`\DCohxRefSystemName` When using `\DCohx`, the macro `\DCohxRefSystemName` is invoked to produce the reference system name corresponding to the input system name. By default, this is a $R.$ symbol with subscript the input system name. You may redefine this macro if you prefer another reference system name:

```
\renewcommand\DCohxRefSystemName[1]{E_{#1}}
```

Then: `\DCohx{\rho}{X}{X'}{\Gamma_X}{\Gamma_{X'}}` → $\bar{D}_{X\to X'}(\rho_{X'E_X} \,\|\, \Gamma_X, \Gamma_{X'})$

`\DCSym` The symbol to use to denote a coherent relative entropy. You may redefine this command to change the symbol. (This works like `\HSym` and `\DSym` above.)

`\DCoh` Typeset a coherent relative entropy using the old notation. The syntax is:

`\DCoh[`⟨*epsilon*⟩`]`⟨*size-spec*⟩`{`⟨*rho*⟩`}{`⟨*R*⟩`}{`⟨*X'*⟩`}{`⟨Γ_R⟩`}{`⟨$\Gamma_{X'}$⟩`}`

For example, `\DCoh[\epsilon]{\rho}{R}{X'}{\Gamma_R}{\Gamma_{X'}}` gives $\bar{D}^{\epsilon}_{R\to X'}(\rho_{X'R} \,\|\, \Gamma_R, \Gamma_{X'})$.

The subscript $X'R$ (or whatever the system names) is automatically added to the ⟨*rho*⟩ argument. If this is not desired, then begin the ⟨*rho*⟩ argument with a star. For example, `\DCoh{*\sigma_R\otimes\rho_{X'}}{R}{X'}{\Gamma_R}{\Gamma_{X'}}` gives $\bar{D}_{R\to X'}(\sigma_R \otimes \rho_{X'} \,\|\, \Gamma_R, \Gamma_{X'})$.

The ⟨*size-spec*⟩ is of course optional and follows the same syntax as everywhere else (subsection 2.2).

The command `\DCoh` is defined only if the package option `qitobjdef=stdset` is set (which is the default).

Appearance and alternative notation

You may change the notation of any of the above relative entropy measures by redefining the corresponding commands as follows:

```
\renewcommand{\DCoh}{\DCohbase{\tilde\DSym}}
```

Then: `\DCoh[\epsilon]{\rho}{R}{X'}{\Gamma_R}{\Gamma_{X'}}` → $\boxed{\tilde{D}^{\epsilon}_{R\to X'}(\rho_{X'R} \,\|\, \Gamma_R, \Gamma_{X'})}$.

Base relative entropy command

As for the other entropy measures, there is a “base coherent relative entropy command” `\DCohbase`. Its syntax is:

`\DCohbase{`⟨*D-symbol*⟩`}[`⟨*epsilon*⟩`]`⟨*size-spec*⟩`{`⟨*rho*⟩`}{`⟨*R*⟩`}{`⟨*X'*⟩`}{`⟨Γ_R⟩`}{`⟨$\Gamma_{X'}$⟩`}`

See also the implementation documentation below for more specific information on how to customize parts of the rendering, for instance.

6.5 Custom Qit Objects

Changed in v2.0 [2017/06/17]: Introduced the Qit Objects infrastructure.

You can create your own Qit Object Implementation as follows. You need two components: a *Parse* macro and a *Render* macro.

The *Parse* macro is responsible for parsing input LaTeX tokens as necessary, and building an argument list (which will be passed on to the *Render* macro).

`\qitobjAddArg`
`\qitobjAddArgx`

The *Parse* macro (or any helper macro it calls) should call `\qitobjAddArg` to add arguments for the eventual call to *Render*. The `\qitobjAddArg` macro does not expand its argument. The `\qitobjAddArgx` works like `\qitobjAddArg`, but it accepts a single LaTeX command as its only argument, expands it, and adds the contents as a single new argument for the renderer.

`\qitobjParseDone`

Once the parser is finished, it must call `\qitobjParseDone`.

The *Render* macro is responsible for displaying the final Qit Object. It should accept mandatory arguments in the exact number as there were calls to `\qitobjAddArg`/`\qitobjAddArgx`.

`\qitobjDone`

The *Render* macro must call `\qitobjDone` after it is finished, to do some cleaning up and to close the local LaTeX group generated by the Qit Ojbect infrastructure.

`\DefineQitObject`

Declare your new Qit Object using the `\DefineQitObject` macro, using the syntax `\DefineQitObject{`⟨*name*⟩`}{`⟨*ParseCommand*⟩`}{`⟨*RenderCommand*⟩`}`. This declares the command `\`⟨*name*⟩ as your Qit Object.

You may define different Qit Objects (using different names) recycling the same parsers/renderers if needed. For instance, `\Hfnbase` uses the same renderer as `\Hbase`.

`\DefineTunedQitObject`

The `\DefineTunedQitObject` macro is a bit more powerful. It allows you to specify some fixed initial arguments to the parser, as well as to provide some local definitions which are in effect only during parsing and rendering of the Qit Object. This is useful if you would like to declare an alternative type of Qit Object to an existing one, where you just change some aspect of the behavior of the original Qit Object.

Usage: `\DefineTunedQitObject{`⟨*name*⟩`}{`⟨*parse command*⟩`}{`⟨*render command*⟩`}{`⟨*fixed first argument(s)*⟩`}{`⟨*custom definitions*⟩`}`

The `{`⟨*first fixed argument(s)*⟩`}` must be a single argument, i.e., a single LaTeX group, which may contain several arguments, for instance: `{{A}{B}}`.

For instance, `\DCohx` is defined, using the same parser and renderer as for `\DCoh`, as follows:

```
\def\DCohxRefSystemName#1{R_{#1}}
\def\DCohxStateSubscripts#1#2{#2\DCohxRefSystemName{#1}}
```

```
\DefineTunedQitObject{DCohx}{\DCohbaseParse}{\DCohbaseRender}%
{{\DCSym}}% initial args
{\let\DCohbaseStateSubscripts\DCohxStateSubscripts}% local defs
```

Useful helpers

There are some useful helpers for both the *Parse* and *Render* macros. More extensive documentation is available in the “Implementation” section below.

`\phfqit@parse@sizesarg` Parse a ⟨*size-spec*⟩ optional argument.

`\phfqitParen`
`\phfqitSquareBrackets`
`\phfqitCurlyBrackets` Produce a parenthetic expression (or square or curly brackets) with the appropriate size and with the given contents.

Example

Here is a simple example: let’s build a work cost of transition Qit Object to display something like “$W(\sigma \to \rho)$.”

The arguments to be given are: they are $\langle\sigma\rangle$ and $\langle\rho\rangle$. We would also like to accept an optional size specification ⟨*size-spec*⟩. We should decide on a convenient syntax to specify them. Here, we’ll settle for simply `\WorkCostTransition‘\Big{\rho}{\sigma}`.

We can now write the *Parse* macro. We use the `\phfqit@parsesizearg` helper, which stores the optional ⟨*size-spec*⟩ into the `\phfqit@val@sizearg` macro before deferring our second helper macro. We then add arguments (for an eventual call to the *Render* macro) using `\qitobjAddArg` (or `\qitobjAddArgx`).

```
\makeatletter
\newcommand\WorkCostTransitionParse{%
  \phfqit@parsesizearg\WorkCostTransitionParse@%
}
% Helper to parse further input arguments:
\newcommand\WorkCostTransitionParse@[2]{% {\rho}{\sigma}
  \qitobjAddArgx\phfqit@val@sizearg% size arg
  \qitobjAddArg{#1}% rho
  \qitobjAddArg{#2}% sigma
  \qitobjParseDone%
}
\makeatother
```

The render macro should simply display the quantity, with the arguments given as usual mandatory arguments. We invoke the `\phfqitParens` helper, which produces the parenthesis at the correct size given the size spec tokens.

```
\newcommand\WorkCostTransitionRender[3]{% {size-spec-tokens}{\rho}{\sigma}
  W\phfqitParens#1{#2 \to #3}%
   \qitobjDone
}
```

Now declare the Qit Object:

```
\DefineQitObject{WorkCostTransition}{\WorkCostTransitionParse}{\WorkCostTransitionRender}
```

Then: `` \WorkCostTransition`\Big{\rho}{\sigma} `` $\to \boxed{W\Big(\rho \to \sigma\Big)}$

You might want to check out the implementations of `\HbaseParse` and `\HbaseRender`, or `\DbaseParse` and `\DbaseRender` if you'd like to see some more involved examples.

7 Implementation

First, load dependent packages. Toolboxes, fonts and so on.

```
\RequirePackage{calc}
\RequirePackage{etoolbox}
\RequirePackage{amsmath}
\RequirePackage{amssymb}
\RequirePackage{dsfont}
\RequirePackage{mathrsfs}
\RequirePackage{mathtools}
```

Package xparse is needed in order to get paren matching right for `\Hfn`.

```
\RequirePackage{xparse}
```

Package options are handled via xkeyval & kvoptions (see implementation doc for phfnote).

```
\RequirePackage{xkeyval}
\RequirePackage{kvoptions}
```

7.1 Simple Symbols and Shorthands

7.1.1 General Symbols

These symbols are documented in section 3.

`\Hs` Hilbert space.

```
\newcommand{\Hs}{\mathscr{H}}
```

`\Ident` Identity operator, $\mathds{1}$.

```
\newcommand{\Ident}{\mathds{1}}
```

`\IdentProc` Identity process.

TODO: this could be implemented as a Qit Object.

```
\def\IdentProc{%
  \phfqit@parsesizearg\phfqit@IdentProc@maybeA%
}
\newcommand\phfqit@IdentProc@maybeA[1][]{%
  \def\phfqit@IdentProc@val@A{#1}%
  \phfqit@IdentProc@maybeB%
}
\newcommand\phfqit@IdentProc@maybeB[1][]{%
  \def\phfqit@IdentProc@val@B{#1}%
  \phfqit@IdentProc@arg%
}
\def\phfqit@IdentProc@arg#1{%
  \def\phfqit@IdentProc@val@arg{#1}%
```

At this point, prepare the three arguments, each expanded exactly as they were when given to these macros, and delegate the formatting to `\phfqit@IdentProc@do`.

```
  \edef\@tmp@args{%
    {\expandonce{\phfqit@IdentProc@val@A}}%
    {\expandonce{\phfqit@IdentProc@val@B}}%
    {\expandonce{\phfqit@IdentProc@val@arg}}%
  }%
  \expandafter\phfqit@IdentProc@do\@tmp@args%
}
\def\phfqit@IdentProc@do#1#2#3{%
  \operatorname{id}_{#1\notblank{#2}{\to #2}{}}%
  \notblank{#3}{\expandafter\phfqitParens\phfqit@val@sizearg{#3}}{}%
}
```

`\ee^...` Macro for the exponential.

Because the character ^ might have different catcodes (e.g. with the breqn package), we use a hack with `\detokenize`. Basically this boils down to `\def\ee^#1{\phfqitExpPowerExpression{#1}}` and `\def\phfqitExpPowerExpression#1{e^{#1}}`, up to making sure that all the ^ symbols are compared without catcodes.

Changed in v4.0 [2021/10/07]: Fixed the definition of `\ee` in order to support the case where the catcode of ^ changes.

```
\edef\phfqit@def@hat{\detokenize{^}}
\expandafter\def\expandafter\phfqit@ee@gobblehat\phfqit@def@hat{%
  \phfqitExpPowerExpression}
\def\phfqitExpPowerExpression#1{e^{#1}}
\def\ee#1{\expandafter\phfqit@ee@gobblehat\detokenize{#1}}
\robustify\phfqitExpPowerExpression
\robustify\ee
```

7.1.2 Math Operators

See user documentation in subsection 3.1.

`\tr`
`\supp`
`\rank`
`\linspan`
`\spec`
`\diag`

Some common math operators. Note that `\span` is already defined by LaTeX, so we resort to `\linspan` for the linear span of a set of vectors.

```
\DeclareMathOperator{\tr}{tr}
\DeclareMathOperator{\supp}{supp}
\DeclareMathOperator{\rank}{rank}
\DeclareMathOperator{\linspan}{span}
\DeclareMathOperator{\spec}{spec}
\DeclareMathOperator{\diag}{diag}
```

`\phfqit@Realpart`
`\phfqit@Imagpart`

Provide math operators for Re and Im. The aliasing to the actual commands `\Re` and `\Im` is done later, when we process the package options.

```
\let\phfqit@Re\Re
\DeclareMathOperator{\phfqit@Realpart}{Re}%
\let\phfqit@Im\Im
\DeclareMathOperator{\phfqit@Imagpart}{Im}%
```

7.1.3 Poly

`\poly`

Poly symbol.

```
\DeclareMathOperator{\poly}{poly}
```

7.1.4 Bits and Bit Strings

See documentation in subsection 3.3

`\bit`
`\bitstring`

Bits and bit strings.

```
\newcommand\bit[1]{\texttt{#1}}
\newcommand\bitstring[1]{\phfqit@bitstring{#1}}
```

The implementation of `\bitstring` needs some auxiliary internal macros.

```
\def\phfqit@bitstring#1{%
  \begingroup%
  \setlength{\phfqit@len@bit}{\maxof{\widthof{\bit{0}}}{\widthof{\bit{1}}}}%
  \phfqitBitstringFormat{\phfqit@bitstring@#1\phfqit@END}%
  \endgroup%
}
```

The internal `\phfqit@bitstring@` macro picks up the next bit, and puts it into a LaTeX `\makebox` on its own with a fixed width.

```
\def\phfqit@bitstring@#1#2\phfqit@END{%
  \makebox[\phfqit@len@bit][c]{\phfqitBitstringFormatBit{#1}}%
  \if\relax\detokenize\expandafter{#2}\relax%
  \else%
```

If there are bits left, then recurse for the rest of the bitstring:

```
    \phfqitBitstringSep\phfqit@bitstring@#2\phfqit@END%
  \fi%
}
\newlength\phfqit@len@bit
```

`\phfqitBitstringSep`
`\phfqitBitstringFormat`

Redefine these to customize the bit string appearance.

```
\newcommand\phfqitBitstringSep{\hspace{0.3ex}}
\newcommand\phfqitBitstringFormat[1]{\ensuremath{\underline{\overline{#1}}}}
\def\phfqitBitstringFormatBit{\bit}
```

7.1.5 Logical Gates

See user documentation in subsection 3.4.

`\gate` Generic macro to format a gate name.

```
\DeclareRobustCommand\gate[1]{\ifmmode\textsc{\lowercase{#1}}%
  \else{\rmfamily\textsc{\lowercase{#1}}}\fi}
```

`\AND`
`\XOR`
`\CNOT`
`\NOT`
`\NOOP`

Some common gates.

```
\newcommand{\AND}{\gate{And}}
\newcommand{\XOR}{\gate{Xor}}
\newcommand{\CNOT}{\gate{C-Not}}
\newcommand{\NOT}{\gate{Not}}
\newcommand{\NOOP}{\gate{No-Op}}
```

7.1.6 Lie Groups & Algebras

`\uu(N)`
`\UU(N)`
`\su(N)`
`\SU(N)`
`\so(N)`
`\SO(N)`
`\slalg(N)`
`\SL(N)`
`\GL(N)`
`\SN(N)`

Some Lie Groups & Algebras. See section 4

```
\def\uu(#1){\phfqitLieAlgebra{u}{#1}}
\def\UU(#1){\phfqitLieGroup{U}{#1}}
\def\su(#1){\phfqitLieAlgebra{su}{#1}}
\def\SU(#1){\phfqitLieGroup{SU}{#1}}
\def\so(#1){\phfqitLieAlgebra{so}{#1}}
\def\SO(#1){\phfqitLieGroup{SO}{#1}}
\def\slalg(#1){\phfqitLieAlgebra{sl}{#1}} % \sl is "slanted font" in TeX
\def\SL(#1){\phfqitLieGroup{SL}{#1}}
\def\GL(#1){\phfqitLieGroup{GL}{#1}}
\def\SN(#1){\phfqitDiscreteGroup{S}{#1}}
```

`\phfqitLieAlgebra`
`\phfqitLieGroup`
`\phfqitDiscreteGroup`

Override these to change the appearance of the group names or algebra names. The first argument is the name of the group or algebra (e.g. su or SU), and the second argument is the parenthesized argument e.g. “N”.

```
\newcommand\phfqitLieAlgebra[2]{\mathfrak{#1}({#2})}
\newcommand\phfqitLieGroup[2]{\mathrm{#1}({#2})}
\newcommand\phfqitDiscreteGroup[2]{\mathrm{#1}_{#2}}
```

7.2 Helper for parsing a size argument

`\phfqit@parsesizearg`

Utility to parse size argument with the backtick specification (subsection 2.2).

Parses a size argument, if any, and stores it into `\phfqit@val@sizearg`. The value stored can directly be expanded as an optional argument to a `\DeclarePairedDelimiter`-compatible command (see mathtools package).

`#1` should be a command token. It is the next action to take, after argument has been parsed.

```
\def\phfqit@parsesizearg#1{%
  \begingroup%
  \mathcode`\`="0060\relax%
  \gdef\phfqit@val@sizearg{}%
  \def\phfqit@tmp@contwithsize{\phfqit@parsesizearg@withsize{#1}}%
  \@ifnextchar`{\phfqit@tmp@contwithsize}{\endgroup#1}%
}
\def\phfqit@parsesizearg@withsize#1`#2{%
  \def\phfqit@tmp@x{#2}%
  \def\phfqit@tmp@star{*}%
  \ifx\phfqit@tmp@x\phfqit@tmp@star%
    \gdef\phfqit@val@sizearg{*}%
    \def\phfqit@tmp@cont{\endgroup#1}%
    \expandafter\phfqit@tmp@cont%
  \else%
    \gdef\phfqit@val@sizearg{[#2]}%
    \def\phfqit@tmp@cont{\endgroup#1}%
    \expandafter\phfqit@tmp@cont%
  \fi%
}
```

`\phfqitDeclarePairedDelimiterXWithAltSizing`
`\phfqitDeclarePairedDelimiterXPPWithAltSizing`

Macros that behave exactly like mathtools' `\DeclarePairedDelimiterX` and `\DeclarePairedDelimiterXPP`, except that their size argument can also be specified by a backtick as for entropy measures or other objects in this package (e.g., to define `\abs` such that ``$\abs`\Big{x - y}$`` gives $\Big|x - y\Big|$).

Changed in v4.0 [2021/10/07]: Added `\phfqitDeclarePairedDelimiterXWithAltSizing` and `\phfqitDeclarePairedDelimiterXPPWithAltSizing`.

```
\def\phfqitDeclarePairedDelimiterXWithAltSizing{%
  \phfqitDeclareMathtoolsPairedDelimiterCmdWithAltSizing\DeclarePairedDelimiterX
}
\def\phfqitDeclarePairedDelimiterXPPWithAltSizing{%
  \phfqitDeclareMathtoolsPairedDelimiterCmdWithAltSizing\DeclarePairedDelimiterXPP
}
\def\phfqitDeclareMathtoolsPairedDelimiterCmdWithAltSizing#1#2{%
  \begingroup
    \escapechar=-1\relax
    \xdef\phfqit@tmp@thecmd{%
      \expandafter\noexpand\csname phfqit@paireddelim@def@\string#2\endcsname}%
  \endgroup
  \edef\x{%
    \noexpand\phfqit@paireddelim@parsesizearg{\expandonce\phfqit@tmp@thecmd}%
  }%
  \expandafter\DeclareRobustCommand\expandafter#2\expandafter{\x}%
  \expandafter#1\phfqit@tmp@thecmd
}
\def\phfqit@paireddelim@parsesizearg#1{%
  \phfqit@parsesizearg{\expandafter#1\phfqit@val@sizearg}%
}
```

7.3 Bra-Ket Notation

`\phfqitKetsBarSpace`
`\phfqitKetsRLAngleSpace`

These macros can be redefined to fine-tune the space that is inserted in some of the ket/bra constructs.

`\phfqitKetsBarSpace` is the space around the vertical bars, e.g., in a bra-ket, or in matrix elements. (Previously, this space was hard-coded to `\hspace*{0.2ex}`. Now, the spacing can be specified in this macro. We furthermore use "math units" (mu), which are more suitable for specifying space in math mode; recall that $1\,\mathrm{mu}$ is $1/18\,\mathrm{em}$ in the math font.)

```
\def\phfqitKetsBarSpace{\mkern 1.5mu\relax}
```

The macro `\phfqitKetsRLAngleSpace` specifies the space between the right angle bracket and the left angle bracket in ket-bra type constructs ($|\phi\rangle\langle\psi|$). By default, it expands to negative space to bring the angle brackets closer together. (Previously, this space was hard-coded to `\hspace*{-0.25ex}`.)

```
\def\phfqitKetsRLAngleSpace{\mkern -1.8mu\relax}
```

\ket
\bra
\braket
\ketbra
\proj
\matrixel
\dmatrixel

Bras, kets, norms, some delimiter stuff. User documentation in subsection 5.1.

```
\phfqitDeclarePairedDelimiterXWithAltSizing\ket[1]{\lvert}{\rangle}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\bra[1]{\langle}{\rvert}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\braket[2]{\langle}{\rangle}{%
  {#1}\phfqitKetsBarSpace\delimsize\vert\phfqitKetsBarSpace{#2}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\ketbra[2]{\lvert}{\rvert}{%
  {#1}\delimsize\rangle\phfqitKetsRLAngleSpace\delimsize\langle{#2}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\proj[1]{\lvert}{\rvert}{%
  {#1}\delimsize\rangle\phfqitKetsRLAngleSpace\delimsize\langle{#1}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\matrixel[3]{\langle}{\rangle}{%
  {#1}\phfqitKetsBarSpace\delimsize\vert\phfqitKetsBarSpace{#2}%
  \phfqitKetsBarSpace\delimsize\vert\phfqitKetsBarSpace{#3}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\dmatrixel[2]{\langle}{\rangle}{%
  {#1}\phfqitKetsBarSpace\delimsize\vert\phfqitKetsBarSpace{#2}%
  \phfqitKetsBarSpace\delimsize\vert\phfqitKetsBarSpace{#1}%
}
```

\phfqitKetsBeforeCommaSpace
\phfqitKetsAfterCommaSpace
\innerprod

We also provide the `\innerprod` macro at this point. Customize the inner spacing before and after the comma with `\phfqitKetsBeforeCommaSpace` and `\phfqitKetsAfterCommaSpace`.

```
\def\phfqitKetsBeforeCommaSpace{}
\def\phfqitKetsAfterCommaSpace{\mkern 1.5mu\relax}
\phfqitDeclarePairedDelimiterXWithAltSizing\innerprod[2]{\langle}{\rangle}{%
  {#1}\phfqitKetsBeforeCommaSpace,\phfqitKetsAfterCommaSpace{#2}%
}
```

\phfqitOKetsBarSpace
\phfqitOKetsRLAngleSpace

These macros are the same as `\phfqitKetsBarSpace` and `\phfqitKetsRLAngleSpace`, except that they specify the corresponding spacing for the `\oket` family of bra/ket-type constructs.

```
\def\phfqitOKetsBarSpace{\phfqitKetsBarSpace}
\def\phfqitOKetsRLAngleSpace{\phfqitKetsRLAngleSpace}
```

`\oket` `\obra` `\obraket` `\oketbra` `\oproj` `\omatrixel` `\odmatrixel`

Again Bras, kets, but for operator space this time. User documentation in subsection 5.1. These definitions depend on `\llangle` and `\rrangle` being available and expanding to valid delimiter symbols. See also the `llanglefrommnsymbolfonts` option.

Changed in v4.1 [2021/10/08]: Added support for `\oket` and friends.

```
\phfqitDeclarePairedDelimiterXWithAltSizing\oket[1]{\lvert}{\rrangle}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\obra[1]{\llangle}{\rvert}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\obraket[2]{\llangle}{\rrangle}{%
  {#1}\phfqitOKetsBarSpace\delimsize\vert\phfqitOKetsBarSpace{#2}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\oketbra[2]{\lvert}{\rvert}{%
  {#1}\delimsize\rrangle\phfqitOKetsRLAngleSpace\delimsize\llangle{#2}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\oproj[1]{\lvert}{\rvert}{%
  {#1}\delimsize\rrangle\phfqitOKetsRLAngleSpace\delimsize\llangle{#1}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\omatrixel[3]{\llangle}{\rrangle}{%
  {#1}\phfqitOKetsBarSpace\delimsize\vert\phfqitOKetsBarSpace{#2}%
  \phfqitOKetsBarSpace\delimsize\vert\phfqitOKetsBarSpace{#3}%
}
\phfqitDeclarePairedDelimiterXWithAltSizing\odmatrixel[2]{\llangle}{\rrangle}{%
  {#1}\phfqitOKetsBarSpace\delimsize\vert\phfqitOKetsBarSpace{#2}%
  \phfqitOKetsBarSpace\delimsize\vert\phfqitOKetsBarSpace{#1}%
}
```

7.4 Delimited Expressions

Delimited expressions are documented in subsection 5.2.

`\abs` `\avg` `\norm`

Other delimited expressions.

```
\phfqitDeclarePairedDelimiterXWithAltSizing\abs[1]{\lvert}{\rvert}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\avg[1]{\langle}{\rangle}{{#1}}
\phfqitDeclarePairedDelimiterXWithAltSizing\norm[1]{\lVert}{\rVert}{{#1}}
```

`\phfqitDefineNorm`

Use `\phfqitDefineNorm\opnorm{}{_\infty}` to define specific norms, with the syntax `\phfqitDefineNorm`⟨*new command name*⟩⟨*tokens before*⟩⟨*tokens after*⟩. If you need arguments in the before/after tokens, then your norm starts to be more complicated than what `\phfqitDefineNorm` can handle, perhaps it's best you use `\phfqitDeclarePairedDelimiterXPPWithAltSizing` directly.

Changed in v4.0 [2021/10/07]: Added `\phfqitDefineNorm`.

```
\def\phfqitDefineNorm#1#2#3{%
  \phfqitDeclarePairedDelimiterXPPWithAltSizing#1[1]{#2}{\lVert}{\rVert}{#3}{{##1}}%
}
```

`\phfqit@insideinterval` Format the contents of an interval. Utility for defining `\intervalc` and friends.

```
\def\phfqit@insideinterval#1#2{{#1\mathclose{},\mathopen{}#2}}
```

`\intervalc`
`\intervalo`
`\intervalco`
`\intervaloc`

Open/Closed/Semi-Open Intervals

```
\phfqitDeclarePairedDelimiterXWithAltSizing\intervalc[2]{[}{]}{%
  \phfqit@insideinterval{#1}{#2}}
\phfqitDeclarePairedDelimiterXWithAltSizing\intervalo[2]{]}{[}{%
  \phfqit@insideinterval{#1}{#2}}
\phfqitDeclarePairedDelimiterXWithAltSizing\intervalco[2]{[}{[}{%
  \phfqit@insideinterval{#1}{#2}}
\phfqitDeclarePairedDelimiterXWithAltSizing\intervaloc[2]{]}{]}{%
  \phfqit@insideinterval{#1}{#2}}
```

7.5 Entropy Measures and Other Qit Objects

Changed in v2.0 [2017/06/17]: Introduced the Qit Objects infrastructure.

7.5.1 Some Internal Utilities

`\phfqitParens` Simple parenthesis-delimited expression, with `\DeclarePairedDelimiterX`-compatible syntax. For example,

`\phfqitParens{`⟨*content*⟩`}` → `(` ⟨*content*⟩ `)`

`\phfqitParens*{`⟨*content*⟩`}` → `\left(` ⟨*content*⟩ `\right)`

`\phfqitParens[\big]{`⟨*content*⟩`}` → `\bigl(` ⟨*content*⟩ `\bigr)`

```
\DeclarePairedDelimiterX\phfqitParens[1]{(}{)}{#1}
```

`\phfqitSquareBrackets` Simple bracket-delimited expression, with `\DeclarePairedDelimiterX`-compatible syntax. For example,

`\phfqitSquareBrackets{`⟨*content*⟩`}` → `[` ⟨*content*⟩ `]`

`\phfqitSquareBrackets*{`⟨*content*⟩`}` → `\left[` ⟨*content*⟩ `\right]`

`\phfqitSquareBrackets[\big]{`⟨*content*⟩`}` → `\bigl[` ⟨*content*⟩ `\bigr]`

```
\DeclarePairedDelimiterX\phfqitSquareBrackets[1]{[}{]}{#1}
```

`\phfqitCurlyBrackets` Simple bracket-delimited expression, with `\DeclarePairedDelimiterX`-compatible syntax. For example,

`\phfqitSquareBrackets{`⟨*content*⟩`}` → `\{` ⟨*content*⟩ `\}`

`\phfqitSquareBrackets*{`⟨*content*⟩`}` → `\left\{` ⟨*content*⟩ `\right\}`

`\phfqitSquareBrackets[\big]{`⟨*content*⟩`}` → `\bigl\{` ⟨*content*⟩ `\bigr\}`

```
\DeclarePairedDelimiterX\phfqitCurlyBrackets[1]{\{}{\}}{#1}
```

7.5.2 Machinery for Qit Objects

See also user documentation in subsection 6.5.

`\QitObject` The argument is the entropic quantity type or object kind (or “entropic quantity driver”): one of `Hbase`, `Hfnbase`, `Dbase`, `DCbase`, or any other custom object.

```
\newcommand\QitObject[1]{%
  \begingroup%
    \preto\QitObjectDone{\endgroup}%
    \QitObjectInit%
    \csname QitObj@reg@#1@initdefs\endcsname%
%%\message{DEBUG: \detokenize{\QitObject{#1}}}%
    \def\QitObj@args{}%
    \def\qitobjParseDone{\QitObj@proceedToRender{#1}}%
    \def\qitobjDone{\QitObjectDone}%
    \csname QitObj@reg@#1@parse\endcsname%
}
```

`\DefineQitObject` `\DefineTunedQitObject` Define a new Qit Object implementation with this macro. A Qit Object implementation is specified in its simplest form by a *name*, a *Parser* and a *Renderer* (a single LaTeX macro each). The more advanced `\DefineTunedQitObject` allows you in addition to specify local definitions to override defaults, as well as some initial arguments to the parser.

```
\def\DefineQitObject#1#2#3{%
  \DefineTunedQitObject{#1}{#2}{#3}{}{}%
}%
\def\DefineTunedQitObject#1#2#3#4#5{%
  \csdef{#1}{\QitObject{#1}#4}%
  \expandafter\robustify\csname #1\endcsname%
  \cslet{QitObj@reg@#1@parse}#2%
  \cslet{QitObj@reg@#1@render}#3%
  \csdef{QitObj@reg@#1@initdefs}{#5}%
}
```

Here are some callbacks meant for Qit Object implementations (“types”/“drivers”).

`\qitobjAddArg` `\qitobjAddArgx` These macros should only be called from within a *Parse* macro of a qit object type. Append an argument in preparation for an eventual call to the corresponding

Render macro. `\qitobjAddArg` does not expand its contents. `\qitobjAddArgx` expects a single command name as argument; it expands the command once and stores those tokens as a single new argument.

```
\def\qitobjAddArg#1{%
  \appto\QitObj@args{{#1}}%
}
\def\qitobjAddArgx#1{%
  \expandafter\qitobjAddArg\expandafter{#1}%
}
```

`\qitobjParseDone`
`\qitobjDone`

These macros MUST be called at the end of the respective *Parse* (`\qitobjParseDone`) and *Render* (`\qitobjDone`) implementations (otherwise processing doesn't proceed, LaTeX groups won't be closed, and it will be a mess).

These macros are correctly defined in `\QitObject` actually. Here we provide empty definitions so that the *Render* and *Parse* user implementation macros can be called stand-alone, too.

```
\def\qitobjParseDone{}
\def\qitobjDone{}
```

`\QitObjectDone`

A hook which gets called after a Qit Object is displayed. This should really stay empty on the global scope. However you can locally append or prepend to it in tuned definitions for `\DeclareTunedQitObject` to perform additional actions at the end of the Qit Object, for instance to close an additional LaTeX group.

```
\def\QitObjectDone{}
```

`\QitObjectInit`

A hook which gets called before the parsing phase of a Qit Object. This should really stay empty on the global scope. However you can locally append or prepend to it in tuned definitions for `\DeclareTunedQitObject` to perform additional actions before parsing the Qit Object (but which have to be made within the LaTeX group of the Qit Object). You can use this to prepend code to `\QitObjectDone` so that you code gets called *before* the inner LaTeX group is closed.

```
\def\QitObjectInit{}
```

An internal helper; it's useful to keep it separate for readability and for debugging.

```
\def\QitObj@proceedToRender#1{%
%%\message{DEBUG: Rendering #1|\detokenize\expandafter{\QitObj@args}|}%
  \expandafter\def\expandafter\x\expandafter{%
    \csname QitObj@reg@#1@render\endcsname}%
  \expandafter\x\QitObj@args%
}
```

7.5.3 Qit Object Implementation: Entropy, Conditional Entropy

See also the user doc in subsection 6.1.

`\HbaseParse` Base parser macro for usual entropy measures; possibly conditional and/or smooth.

USAGE: `\Hbase{`⟨*H-symbol*⟩`}{`⟨*subscript*⟩`}`⟨*size-spec*⟩`[`⟨*state*⟩`][`⟨*epsilon*⟩`]{`⟨*target system*⟩`}[`⟨*conditioning system*⟩`]`

The argument ⟨*size-spec*⟩ is optional, and is documented in subsection 2.2. For example ⟨*size-spec*⟩ = `‘*` or `‘\Big`.

Examples:

`\Hbase{\hat{H}}{\mathrm{max}}[\rho][\epsilon]{E}[X']` → $\boxed{\hat{H}^{\epsilon}_{\mathrm{max}}(E\,|\,X')_{\rho}}$

`\Hbase{\hat{H}}{\mathrm{max}}‘*[\rho][\epsilon]{\bigotimes_i E}[X']` → $\boxed{\hat{H}^{\epsilon}_{\mathrm{max}}\left(\bigotimes_i E\,\middle|\,X'\right)_{\rho}}$

The `\HbaseParse` macro is responsible for parsing the arguments to `\Hbase`. We should parse the arguments using helper macros as needed, adding rendering arguments with `\qitobjAddArg` or `\qitobjAddArgx`, and then calling `\qitobjParseDone`. The arguments are then automatically provided as arguments to the `\HbaseRender` function. We just have to make sure we add the correct number of arguments in the correct order.

```
\def\HbaseParse#1#2{%
```

The first arguments are the mandatory arguments `{`⟨*H-symbol*⟩`}{`⟨*subscript*⟩`}`. Then defer to helper macros for the rest of the parsing.

```
  \qitobjAddArg{#1}%
  \qitobjAddArg{#2}%
  \phfqit@parsesizearg\HbaseParse@%
}
```

Store the delimiter size argument which `\phfqit@parsesizearg` has stored into `\phfqit@val@sizearg`, then parse an optional `[`⟨*state*⟩`]` argument.

```
\newcommand\HbaseParse@[1][]{%
  \qitobjAddArgx{\phfqit@val@sizearg}%
  \qitobjAddArg{#1}%
  \HbaseParse@@%
}
```

Then parse an optional `[`⟨*epsilon*⟩`]` argument, as well as a mandatory `{`⟨*target system*⟩`}` argument.

```
\newcommand\HbaseParse@@[2][]{%
  \qitobjAddArg{#1}%
  \qitobjAddArg{#2}%
  \HbaseParse@@@%
}
```

Finally, parse an optional `[`⟨*conditioning system*⟩`]`.

```
\newcommand\HbaseParse@@@[1][]{%
  \qitobjAddArg{#1}%
  \qitobjParseDone%
}
```

`\HbaseRender` Render the entropy measure.

#1 = "H" symbol to use (e.g. `H`)

#2 = subscript (type of entropy, e.g. `\marthrm{min}`, `0`)

#3 = possible size argument to expand in front of parens command (one of *(empty)*, `*`, or `[\big]` using a standard sizing command)

#4 = the state (e.g. `\rho`), may be left empty

#5 = epsilon argument (superscript to entropy measure), if any, or leave argument empty

#6 = system to measure entropy of

#7 = conditioning system, if any, or else leave the argument empty

```
\def\HbaseRender#1#2#3#4#5#6#7{%
%%\message{DEBUG: HbaseRender\detokenize{{#1}{#2}{#3}{#4}{#5}{#6}{#7}}}%
```

Start with the entropy symbol ('H'), the subscript, and the superscript:

```
  \HbaseRenderSym{#1}_{\HbaseRenderSub{#2}}^{\HbaseRenderSup{#5}}
```

Render the contents of the entropy (parenthetic expression with system & conditioning system), only if the system or conditioning system or state are not empty:

```
  \notblank{#4#6#7}{%
    \HbaseRenderContents{#3}{#6}{#7}%
```

Finally, add the state as subscript, if any:

```
    \HbaseRenderTail{#4}%
  }{}%
```

We're done.

```
  \qitobjDone%
}
```

`\HbaseRenderSym`
`\HbaseRenderSub`
`\HbaseRenderSup`
`\HbaseRenderTail`

Macros to render different parts of the entropy measure. By default, don't do anything special to them (but this might be locally overridden in a tuned Qit Object, for instance).

```
\def\HbaseRenderSym#1{#1}%
\def\HbaseRenderSub#1{#1}%
\def\HbaseRenderSup#1{#1}%
\def\HbaseRenderTail#1{_{#1}}%
```

`\HbaseRenderContents`

For the main contents rendering macro, we need to do a little more work. First, declare a token register in which we will prepare the contents of the parenthetic expression.

```
\newtoks\Hbase@tmp@toks
\def\Hbase@addtoks#1\@Hbase@END@ADD@TOKS{%
  \Hbase@tmp@toks=\expandafter{\the\Hbase@tmp@toks#1}}%
```

Now we need to define the macro which formats the contents of the entropy. The arguments are `#1` = possible sizing argument, `#2` = system name, `#3` = conditioning system if any.

```
\def\HbaseRenderContents#1#2#3{%
```

We need to construct the parenthetic argument to the entropy, which we will store in the token register `\Hbase@tmp@toks`. Start with system name:

```
    \Hbase@tmp@toks={#2}%
```

… add conditional system, if specified:

```
    \notblank{#3}{%
      \Hbase@addtoks\mathclose{}\,\delimsize\vert\,\mathopen{}%
          #3%
          \@Hbase@END@ADD@TOKS%
    }{}%
```

The tokens are ready now. Prepare the argument to the command `\HbaseRenderContentsInnerParens` (normally just `\phfqitParens`), and go:

```
    \edef\tmp@args{\unexpanded{#1}{\the\Hbase@tmp@toks}}%
    \expandafter\HbaseRenderContentsInnerParens\tmp@args%
}
```

`\X`

Macro which expands to the parenthetic expression type macro we would like to use. By default, this is `\phfqitParens`.

```
\def\HbaseRenderContentsInnerParens{\phfqitParens}
```

`\Hbase`

Finally, we declare our base entropic quantity type:

```
\DefineQitObject{Hbase}{\HbaseParse}{\HbaseRender}
```

7.5.4 Qit Object Implementation: Entropy Function

See also the user doc in subsection 6.2.

`\Hfnbase` Base implementation of an entropy function.

Usage: `\Hfnbase{H}{1}{2}(x)` → $H_1^2(x)$, ``\Hfnbase{H}{1}{2}`*(x)`` → $H_1^2(x)$, ``\Hfnbase{H}{1}{2}`\big(x)`` → $H_1^2\big(x\big)$.

We can use the same renderer as `\Hbase`, we just need a different parser. The parser first accepts the mandatory arguments `{`⟨*H-symbol*⟩`}{`⟨*subscript*⟩`}{`⟨*superscript*⟩`}`.

```
\def\HfnbaseParse#1#2#3{%
  \qitobjAddArg{#1}% H-sym
  \qitobjAddArg{#2}% sub
  \phfqit@parsesizearg{\HfnbaseParse@{#3}}%
}
```

Continue to parse a the argument given in parentheses. The first mandatory argument is simply the subscript passed on from the previous macro. It might be tempting to do simply `\def\HfnbaseParse@#1(#2){...}`, but this does not allow for recursive use of parenthesis within the entropy argument, for instance `\Hfn(g(x)+h(y))`. Because of this, we use `xparse`'s `\NewDocumentCommand` which can handle this.

```
\NewDocumentCommand{\HfnbaseParse@}{mr()}{%
  \qitobjAddArgx{\phfqit@val@sizearg}% size-arg
  \qitobjAddArg{}% state
  \qitobjAddArg{#1}% epsilon
  \qitobjAddArg{#2}% system--main arg
  \qitobjAddArg{}% cond system
%%\message{DEBUG: Hfnbase args are |\detokenize\expandafter{\QitObj@args}|}%
  \qitobjParseDone%
}
\DefineQitObject{Hfnbase}{\HfnbaseParse}{\HbaseRender}
```

7.5.5 Qit Object Implementation: Relative Entropy

User documentation in subsection 6.3.

`\DbaseParse` Base macro for relative entropy macros.

USAGE: `\Dbase{`⟨*D-symbol*⟩`}[_`⟨*subscript*⟩`][^`⟨*superscript*⟩`]`⟨*size-spec*⟩`{`⟨*state*⟩`}{`⟨*relative to state*⟩`}`

The subscript and superscripts are optional and don't have to be specified. They may be specified in any order. Repetitions are allowed and concatenates the arguments, e.g., `^{a}_{x}_{y}^{z}_{w}` is the same as `_{xyw}^{az}`.

The ⟨*size-spec*⟩ is a backtick-style specification as always.

```
\def\DbaseParse#1{%
  \qitobjAddArg{#1}% D-sym
  \def\DbaseParse@val@sub{}%
  \def\DbaseParse@val@sup{}%
  \DbaseParse@%
}
\def\DbaseParse@{%
  \@ifnextchar_{\DbaseParse@parsesub}{\DbaseParse@@}%
}
\def\DbaseParse@@{%
  \@ifnextchar^{\DbaseParse@parsesup}{\DbaseParse@@@}%
}
\def\DbaseParse@parsesub_#1{%
  \appto\DbaseParse@val@sub{#1}%
  \DbaseParse@% return to maybe parsing other sub/superscripts
}
\def\DbaseParse@parsesup^#1{%
  \appto\DbaseParse@val@sup{#1}%
  \DbaseParse@% return to maybe parsing other sub/superscripts
}
\def\DbaseParse@@@{%
  \qitobjAddArgx\DbaseParse@val@sub%
  \qitobjAddArgx\DbaseParse@val@sup%
  \phfqit@parsesizearg\DbaseParse@rest%
}
\def\DbaseParse@rest#1#2{%
  \qitobjAddArgx\phfqit@val@sizearg%
  \qitobjAddArg{#1}% rho
  \qitobjAddArg{#2}% Gamma
  \qitobjParseDone%
}
```

`\DbaseRender` Macro which formats a relative entropy of the form $D^{\mathrm{sup}}_{\mathrm{sub}}(A\|B)$:

`\DbaseRender{D}{\mathrm{min}}{\epsilon}{[\big]}{\rho}{\Gamma}`
→ $\boxed{D^{\epsilon}_{\mathrm{min}}\big(\rho\,\big\|\,\Gamma\big)}$

```
\def\DbaseRender#1#2#3#4#5#6{%
%%\message{DEBUG: DbaseRender\detokenize{{#1}{#2}{#3}{#4}{#5}{#6}}}%
```

Start with the entropy symbol (‘H’), the subscript, and the superscript:

```
  \DbaseRenderSym{#1}_{\DbaseRenderSub{#2}}^{\DbaseRenderSup{#3}}
```

Render the contents of the entropy (parenthetic expression with the (one or) two states), only if the arguments are non-empty:

```
  \notblank{#5#6}{%
    \DbaseRenderContents{#4}{#5}{#6}%
  }{}%
```

We're done.

```
  \qitobjDone%
}
```

\DbaseRenderSym
\DbaseRenderSub
\DbaseRenderSup

Macros to render different parts of the entropy measure. By default, don't do anything special to them (but this might be locally overridden in a tuned Qit Object).

```
\def\DbaseRenderSym#1{#1}%
\def\DbaseRenderSub#1{#1}%
\def\DbaseRenderSup#1{#1}%
```

\DbaseRenderContents

Now we need to define the macro which formats the contents of the entropy. First, define a useful token register.

```
\newtoks\Dbase@tmp@toks
\def\Dbase@addtoks#1\@Dbase@END@ADD@TOKS{%
  \Dbase@tmp@toks=\expandafter{\the\Dbase@tmp@toks#1}}%
```

The arguments are #1 = possible sizing argument, #2 = first state, #3 = second state (or operator), if any.

```
\def\DbaseRenderContents#1#2#3{%
```

We need to construct the parenthetic argument to the relative entropy, which we will store in the token register `\Dbase@tmp@toks`. Start with system name:

```
    \Dbase@tmp@toks={#2}%
```

… add conditional system, if specified:

```
    \notblank{#3}{%
      \Dbase@addtoks\mathclose{}\,\delimsize\Vert\,\mathopen{}%
          #3%
          \@Dbase@END@ADD@TOKS%
    }{}%
```

The tokens are ready now. Prepare the argument to the command `\DbaseRenderContentsInnerParens` (by default just `\phfqitParens`), and go:

```
    \edef\tmp@args{\unexpanded{#1}{\the\Dbase@tmp@toks}}%
    \expandafter\DbaseRenderContentsInnerParens\tmp@args%
}
```

\DbaseRenderContentsInnerParens

Macro which expands to the parenthetic expression type macro we would like to use. By default, this is `\phfqitParens`.

```
\def\DbaseRenderContentsInnerParens{\phfqitParens}
```

`\Dbase` Finally, define the `\Dbase` macro by declaring a new qit object.

```
\DefineQitObject{Dbase}{\DbaseParse}{\DbaseRender}
```

7.5.6 Qit Object Type: Coherent Relative Entropy

See also user documentation in subsection 6.4.

`\DCohbaseParse` Base macros for coherent relative entropy-type quantities of the form $\bar{D}^{\epsilon}_{X\to X'}(\rho_{X'R}\Vert\Gamma_X,\Gamma_{X'})$.

USAGE: `\DCohbase{`⟨*D symbol*⟩`}[`⟨*epsilon*⟩`]{`⟨*state or *fully-decorated-state*⟩`}{`⟨*System In*⟩`}{`⟨*System Out*⟩`}{`⟨*Gamma In*⟩`}{`⟨*Gamma Out*⟩`}`

```
\def\DCohbaseParse#1{%
  \qitobjAddArg{#1}% D-sym
  \DCohbaseParse@%
}
\newcommand\DCohbaseParse@[1][]{%
  \qitobjAddArg{#1}% epsilon
  \phfqit@parsesizearg\DCohbaseParse@rest%
}
\def\DCohbaseParse@rest#1#2#3#4#5{%
  % rho, X, X', \Gamma_X, \Gamma_{X'}
  \qitobjAddArgx\phfqit@val@sizearg%
  \DCohbaseParse@parserhosub#1\DCohbaseParse@ENDSTATE{#2}{#3}%
  \qitobjAddArg{#2}%
  \qitobjAddArg{#3}%
  \qitobjAddArg{#4}%
  \qitobjAddArg{#5}%
  \qitobjParseDone%
}
\def\DCohbaseParse@parserhosub{%
  \@ifnextchar*\DCohbaseParse@parserhosub@nosub%
  \DCohbaseParse@parserhosub@wsub%
}
\def\DCohbaseParse@parserhosub@nosub*#1\DCohbaseParse@ENDSTATE#2#3{%
  \qitobjAddArg{#1}% rho
}
\def\DCohbaseParse@parserhosub@wsub#1\DCohbaseParse@ENDSTATE#2#3{%
  \qitobjAddArg{#1_{\begingroup\let\emptysystem\relax%
      \DCohbaseStateSubscripts{#2}{#3}\endgroup}}% all this for "rho" arg
}
```

`\DCohbaseStateSubscripts` Macro which produces the relevant subscript for the state. By default, simply produce “$X'R$” (but don’t produce an “empty system” symbol). This macro may be overridden e.g. locally.

```
\def\DCohbaseStateSubscripts#1#2{%
```

```
  #2#1%
}
```

`\DCohbaseRender` Render the coherent relative entropy.

#1 = “D” symbol

#2 = superscript (epsilon)

#3 = possible size argument tokens (i.e., `[\big]`)

#4 = fully decorated state (i.e., with necessary subscripts as required)

#5 = input system name

#6 = output system name

#7 = Gamma-in

#8 = Gamma-out

```
\def\DCohbaseRender#1#2#3#4#5#6#7#8{%
  %
%%\message{DEBUG: DCohbaseRender here, args are |\detokenize{{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}
  %
  \DCohbaseRenderSym{#1}%
  _{\DCohbaseRenderSystems{#5}{#6}}%
  ^{\DCohbaseRenderSup{#2}}%
  \notblank{#4#7#8}{%
    \DCohbaseRenderContents{#3}{#4}{#7}{#8}%
  }{}%
```

We’re done.

```
  \qitobjDone%
}
```

`\DCohbaseRenderSym` `\DCohbaseRenderSystems` `\DCohbaseRenderSup` Macros to render different parts of the entropy measure. By default, don’t do anything special to them (but this might be locally overridden in a tuned Qit Object)

```
\def\DCohbaseRenderSym#1{#1}%
\def\DCohbaseRenderSystems#1#2{#1\to #2}%
\def\DCohbaseRenderSup#1{#1}%
```

`\DCohbaseRenderContents` Now we define the macro which formats the contents of the entropy.

Define first a useful token register for rendering the contents.

```
\newtoks\DCohbase@tmp@toks
\def\DCohbase@addtoks#1\@DCohbase@END@ADD@TOKS{%
  \DCohbase@tmp@toks=\expandafter{\the\DCohbase@tmp@toks#1}}%
```

The arguments are #1 = possible sizing argument tokens, #2 = decorated state, #3 = Gamma-X, #4 = Gamma-X’.

```
\def\DCohbaseRenderContents#1#2#3#4{%
```

We need to construct the parenthetic argument to the coherent relative entropy, which we will prepare in the token register `\DCohbase@tmp@toks`. Start with the state:

```
    \DCohbase@tmp@toks={#2}%
```

… add conditional system, if specified:

```
    \notblank{#3}{%
      \DCohbase@addtoks\mathclose{}\,\delimsize\Vert\,\mathopen{}%
          #3\@DCohbase@END@ADD@TOKS%
        \notblank{#4}{%
          \DCohbase@addtoks\mathclose{},\mathopen{}%
              #4\@DCohbase@END@ADD@TOKS%
        }{}%
    }{%
        \notblank{#4}{%
          \PackageWarning{phfqit}{Value ‘#4’ ignored because previous parameter
            was blank}%
        }{}%
      }
```

The tokens are ready now. Prepare the argument to the command `\DCohbaseRenderContentsInnerParens` (by default just `\phfqitParens`), and go:

```
    \edef\tmp@args{\unexpanded{#1}{\the\DCohbase@tmp@toks}}%
    \expandafter\DCohbaseRenderContentsInnerParens\tmp@args%
}
```

`\DCohbaseRenderContentsInnerParens`

Macro which expands to the parenthetic expression type macro we would like to use. By default, this is `\phfqitParens`.

```
\def\DCohbaseRenderContentsInnerParens{\phfqitParens}
```

`\DCohbase`

Finally, define the `\DCohbase` macro by declaring a new qit object.

```
\DefineQitObject{DCohbase}{\DCohbaseParse}{\DCohbaseRender}
```

7.6 Additional helpers for entropy measures

`\HSym`

Symbol to use to denote an entropy measure.

```
\def\HSym{H}
```

`\DSym` Symbol to use to denote a relative entropy measure.

```
\newcommand\DSym{D}
```

`\DCSym` Symbol to use for the coherent relative entropy measure.

```
\newcommand\DCSym{\bar\DSym}
```

`\emptysystem` Designates the trivial system (uses symbol for empty set). It is important to this, because of the automatic indexes set on the “rho” argument.

```
\def\emptysystem{\ensuremath{\emptyset}}
```

`\DCohxRefSystemName`
`\DCohxStateSubscripts` Macros helpful for defining `\DCohx`.

```
\def\DCohxRefSystemName#1{R_{#1}}
\def\DCohxStateSubscripts#1#2{#2\DCohxRefSystemName{#1}}
```

Finally, some macros provided for backwards compatibility:

```
\let\@HHbase\Hbase
\let\@DDbase\Dbase
\let\HHSym\HSym
\let\DDSym\DSym
```

7.7 Handle package options

Changed in v2.0 [2017/08/16]: Added the `qitobjdef` package option.

Changed in v2.0 [2017/08/16]: Added the `newReIm` package option.

Initialization code for kvoptions for our package options. See section 2.

```
\SetupKeyvalOptions{
  family=phfqit,
  prefix=phfqit@opt@
}
```

Set of predefined qit objects to load. Either `stdset` (standard set, the default) or `none` (none).

```
\DeclareStringOption[stdset]{qitobjdef}
```

Whether we should load the `\llangle` and `\rrangle` delimiters from the MnSymbol fonts.

```
\DeclareBoolOption[true]{llanglefrommnsymbolfonts}
```

Whether to override LaTeX’s default $\Re$ and $\Im$ symbols by our more readable Re and Im.

```
\DeclareBoolOption[true]{newReIm}
```

Process package options.

```
\ProcessKeyvalOptions*
```

7.7.1 Re/Im symbols

`\Re` `\Im` Provide `\Re` and `\Im` commands to override LaTeX's default if the corresponding package option is set (which is the default).

```
\ifphfqit@opt@newReIm
  \renewcommand{\Re}{\phfqit@Realpart}
  \renewcommand{\Im}{\phfqit@Imagpart}
\fi
```

7.7.2 Load `\llangle` and `\rrangle` from the *MnSymbol* fonts

We need to import `\llangle` and `\rrangle` from the `MnSymbol` fonts.[2]

```
\ifphfqit@opt@llanglefrommnsymbolfonts
  \DeclareFontFamily{OMX}{MnSymbolE}{}
  \DeclareSymbolFont{phfqit@MnLargeSymbols}{OMX}{MnSymbolE}{m}{n}
  \SetSymbolFont{phfqit@MnLargeSymbols}{bold}{OMX}{MnSymbolE}{b}{n}
  \DeclareFontShape{OMX}{MnSymbolE}{m}{n}{
      <-6>  MnSymbolE5
     <6-7>  MnSymbolE6
     <7-8>  MnSymbolE7
     <8-9>  MnSymbolE8
     <9-10> MnSymbolE9
    <10-12> MnSymbolE10
    <12->   MnSymbolE12
  }{}
  \DeclareFontShape{OMX}{MnSymbolE}{b}{n}{
      <-6>  MnSymbolE-Bold5
     <6-7>  MnSymbolE-Bold6
     <7-8>  MnSymbolE-Bold7
     <8-9>  MnSymbolE-Bold8
     <9-10> MnSymbolE-Bold9
    <10-12> MnSymbolE-Bold10
    <12->   MnSymbolE-Bold12
  }{}
  \let\llangle\@undefined
  \let\rrangle\@undefined
  \DeclareMathDelimiter{\llangle}{\mathopen}%
                       {phfqit@MnLargeSymbols}{'164}{phfqit@MnLargeSymbols}{'164}
  \DeclareMathDelimiter{\rrangle}{\mathclose}%
                       {phfqit@MnLargeSymbols}{'171}{phfqit@MnLargeSymbols}{'171}
\fi
```

---

[2] see e.g. *https://tex.stackexchange.com/a/79701/32188*

7.7.3 Standard entropy measures

Load the requested set of qit objects.

```
\def\phfqit@tmp@str@none{none}
\def\phfqit@tmp@str@stdset{stdset}
\ifx\phfqit@opt@qitobjdef\phfqit@tmp@str@none%
```

In this case, do not load any definitions.

```
\else\ifx\phfqit@opt@qitobjdef\phfqit@tmp@str@stdset%
```

In this case, provide our standard set of “qit objects” (i.e., entropy measures).

`\HH` `\Hzero` `\Hmin` `\Hmaxf`

The definition of individual entropy macros just delegates to `\Hbase` with the relevant subscript.

```
\def\HH{\Hbase{\HSym}{}}
\def\Hzero{\Hbase{\HSym}{\mathrm{max},0}}
\def\Hmin{\Hbase{\HSym}{\mathrm{min}}}
\def\Hmaxf{\Hbase{\HSym}{\mathrm{max}}}
\def\Hfn{\Hfnbase{\HSym}{}{}}
\let\Hfunc\Hfn% backwards compatibility
```

`\DD` (Usual) quantum relative entropy. (Actually this is more versatile, because you can also specify subscript and superscript, so you can make on-the-fly custom relative entropy measures.)

```
\def\DD{\Dbase{\DSym}}
```

`\Dminz` “Old” min-relative entropy, based on the Rényi-zero relative entropy.

```
\newcommand\Dminz[1][]{\Dbase{\DSym}_{\mathrm{min,0}}^{#1}}
```

`\Dminf` Min-relative entropy (“new” version).

```
\newcommand\Dminf[1][]{\Dbase{\DSym}_{\mathrm{min}}^{#1}}
```

`\Dmax` Max-relative entropy.

```
\newcommand\Dmax[1][]{\Dbase{\DSym}_{\mathrm{max}}^{#1}}
```

`\Dr` Rob-relative entropy.

```
\newcommand\Dr[1][]{\Dbase{\DSym}_{\mathrm{r}}^{#1}}
```

`\DHyp` Hypothesis testing relative entropy.

```
\newcommand\DHyp[1][\eta]{\Dbase{\DSym}_{\mathrm{H}}^{#1}}
```

`\Dhyp` Hypothesis testing relative entropy (alternative definition).

Changed in v3.1 [2021/07/27]: Added the `\Dhyp` variant of the hypothesis testing relative entropy.

```
\newcommand\Dhyp[1][\eta]{\Dbase{\DSym}_{\mathrm{h}}^{#1}}
```

`\DCoh` Coherent relative entropy (old style).

```
\DefineTunedQitObject{DCoh}{\DCohbaseParse}{\DCohbaseRender}{{\DCSym}}{}
```

`\DCohx` Coherent relative entropy (new style).

```
\DefineTunedQitObject{DCohx}{\DCohbaseParse}{\DCohbaseRender}%
{{\DCSym}}{%
  \let\DCohbaseStateSubscripts\DCohxStateSubscripts%
}
```

End case `qitobjdef=stdset`. Last case is the final `\else` branch which is an error, as we have an unknown set of standard definitions to load.

```
\else
\PackageError{phfqit}{Invalid value ‘\phfqit@opt@qitobjdef’ specified for
  package option ‘qitobjdef’.  Please specify one of ‘stdset’ (the default) or
  ‘none’}{You specified an invalid value to the ‘qitobjdef’ package option of
  the ‘phfqit’ package.}
\fi
\fi
```

Change History

v1.0
  General: Initial version 1
v2.0
  General: Added the `newReIm` package option 3
    Added the `qitobjdef` package option 3
    Introduced the Qit Objects infrastructure 25
    Added the macro `\GL(N)` 6
v3.0
  General: Added the macros `\slalg(n)`, `\SL(N)`, as well as `\phfqitLieAlgebra`, `\phfqitLieGroup`, and `\phfqitDiscreteGroup` . . 6
v3.1
  `\Dhyp`: Added the `\Dhyp` variant of the hypothesis testing relative entropy 40
v4.0
  `\ee^...`: Fixed the definition of `\ee` in order to support the case where the catcode of ^ changes 18
  `\phfqitDeclarePairedDelimiterXPPWithAltSizing`: Added `\phfqitDeclarePairedDelimiterXWithAltSizing` and `\phfqitDeclarePairedDelimiterXPPWithAltSizing` 22
  `\phfqitDefineNorm`: Added `\phfqitDefineNorm` 24
v4.1
  `\odmatrixel`: Added support for `\oket` and friends 24

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.