yquant.sty package documentation

Typesetting quantum circuits in a human-readable language

Benjamin Desef

January 21, 2023

This manual introduces yquant, a LaTeX-only package that outputs quantum circuits. They are entered using a human-readable language that, even from the source code, allows for a fluent understanding of the logic that underlies the circuit. yquant internally builds on TikZ and can be easily combined with arbitrary LaTeX code. Almost one hundred pages of examples complement the formal manual.

Contents

1 Introduction

This document outlines the scope and usage of the yquant package. It contains both a reference and a huge number of examples. yquant is a package that makes typesetting quantum circuits easy; the package is available on CTAN. This beta version 0.7.3 *should* be stable and interfaces are not very likely to change in an incompatible way in the future. Sometimes, backwards-incompatible changes are required or advisable, in which case a compatibility setting will allow to revert back to the old behavior (rather, to maximize compatibility, this is an opt-in setting: unless you choose the new behavior, you will get the old one). Please do report all issues and desirable additions on GitHub.

New in 0.4

1.1 How to read the manual

The probably fastest way to start using yquant is by just scanning through the examples in section 6. A more formal description of the yquant grammar and its fundamental concepts can be found in section 2. If your desire is to change the appearance of yquant elements, use the configuration reference in section 3. The full list of all available gates is provided in section 5. Finally, you may find that yquant *almost* does what you want, but there is some final tweak that you cannot achieve…. Then, have a look at section 4 (or section 1.3).

1.2 Installation

The recommended way of installation is through CTAN. A direct installation from the Git repository to obtain the latest additions and features is be possible by just cloning it to a path visible to your TEX compiler. For example, you may put the source files in the same directory as your document (if you just want to give a try), or you may extract them to `tex/latex/yquant` in your local `texmf` (followed by an update of the file name database). While the repository may contain new additions, they are not thoroughly tested until they end up on CTAN; features that are not documented in this manual are entirely unreliable.

The CTAN repository reflects the most current version tag on Git; the Releases section on GitHub additionally provides a single-file version of the main package, which can for example conveniently be included in arXiv submissions. Note that the arXiv currently provides yquant 0.3.2 out-of-the-box.

New in 0.4

1.3 Purpose of yquant, alternatives

yquant is the acronym for "yet another quantum circuit package." This highlights the fact that nothing that this package provides cannot be achieved by other means. In particular, there are at least the following methods to typeset quantum circuits in TEX.

- Use some external program to draw them and include the output via \includegraphics.
- Use either TEX's own drawing capabilities (the picture environment) or other drawing packages such as TikZ or pstricks.
- Use a package specifically designed to draw quantum circuits (if you feel some other package should be mentioned here, please file an issue):
  - qasm is probably the first of them (in terms of age). It was developed to typeset the circuits found in Nielsen and Chuang's famous *Quantum Computation and Quantum Information* book. qasm consists of a Python 2 script (qasm2circ) that reads a quantum circuit written in a very intuitive language: declare names for your qubits, perform gates on them in each line. qasm2circ converts those circuits into TEX files that internally make use of the xy package to display the output. Consequently, the user is restricted to the set of features that qasm directly offers (which is small). Changes to the output, while possible, will be overwritten if qasm2circ is run again. qasm output often looks sub-optimal do to the fact that, e.g., rectangles are made up of four lines that do not properly connect and give a crumbly general feeling.

    Note that since version 0.3, yquant understands qasm syntax, see section 7.2.

    Maintenance status: last update of qasm in 2005. Also, xy was last updated in 2013, and the script is not compatible out-of-the-box with Python 3, though an automatic conversion should work.
  - qcircuit is probably the most-widely used package. It provides commands that make it much easier to create quantum circuits using the xy package. Its syntax therefore is grid-oriented; inferring what a circuit does or locating a gate in the code can be tough. This is particularly true for multi-qubit gates. Additionally, the \xymatrix syntax is also somewhat cryptic. qcircuit provides some flexibility within the limits of xy as to configuring the output.

Maintenance status: active (GitHub); but remember this is xy based, with last update in 2013.

– quantikz is a relatively recent package that, following the same grid-based approach as qcircuit, instead builds on TikZ as a backend. As a consequence, it provides the full flexibility of customization that TikZ offers, where hardly anything cannot be done. It also reduces burdens of the xy syntax. However, the disadvantages of the grid-based syntax still remain.

  Maintenance status: last update in 2020; the underlying TikZ is actively maintained again by now.

– qpic follows the approach of qasm: It makes use of an external Python program that reads the quantum circuits in an own language and converts them into TikZ commands. The language qpic follows is much more powerful than qasm's. The disadvantage that modifications in the output code will not remain after running the Python script again is mitigated by the possibility to define own TeX macros. Being an external program, qpic's intrinsic set of features (including, e.g., vertically set circuits) are huge. However, the language qpic uses cannot be understood without a detailed study of the manual, it appears to have been designed with the aim to minimize the length of command names. A disadvantage of external programs is that the amount of space gates need is not accessible by the script; hence, manual intervention may be required.

  Maintenance status: last update in 2020; the underlying TikZ is actively maintained, and the script is compatible with Python 3.

1.4 License

2 Basic elements of yquant

yquant, as some of the aforementioned packages, builds on TikZ. Its basic syntax is similar to pgfplots: Start a tikzpicture environment (perhaps passing some options); inside, start a yquant environment.

Inside the yquant environment, TEX will now understand the yquant language—so yquant falls into the same category as qasm and qpic, providing a human-readable language for the specification of the circuit that is not fixed to the actual layout.

However, yquant is a TEX-only package (actually, LATEX 2_ε, but not LATEX3) that requires no external script to run—so it also falls into the same category as qcircuit and quantikz.

Since it runs entirely within TEX, you can at any time interject yquant code with arbitrary TEX or TikZ code (though if it is "too arbitrary," you may need to restart the yquant interpreter).

2.1 General usage

Changed in 0.4

```
% preamble: \usepackage[compat=<version>]{yquant}
\begin{tikzpicture}% tikz options possible
   % tikz commands go here
   \begin{yquant}% yquant options possible. Watch the newlines!
      % yquant and tikz commands go here
   \end{yquant}
   % tikz commands go here
\end{tikzpicture}
```

Changed in 0.4

Note that yquant depends on etoolbox, TikZ, trimspaces, and xkeyval. Additionally, it requires a moderately recent version of LATEX 2_ε, using either LuaLATEX; or (untested) pdfLATEX or XeLATEX.

Optional arguments

The optional arguments for the yquant environment have to appear *on the same line* as the environment itself. If you want to put the arguments into a new line, it is crucial to mask the line break by putting a comment symbol after the environment: `\begin{yquant}%`. Without this comment, yquant will detect your line break (this is one of the few places in TEX where line breaks and spaces are different) and assume that the expression in square brackets instead provides arguments for the following operation!
Finally note that in (non-fragile) beamer frames, this discrimination between

spaces and new lines does not work; the optional arguments will always be counted for the environment, not for the gate. In this case, you can either declare the frame as fragile or (recommended) introduce a blank line between the environment and the options for the first gate.

Compatibility mode New in 0.4

Sometimes, continued development shows that certain choices of interfaces, configuration, or behavior are less ideal than originally thought. In other cases, bugs are detected and fixed. Both may lead to a change in the look of circuits developed with a previous version of `yquant` or even—though this should rarely, if ever, happen, and should be filed as a bug—prevent compilation in the new version.
For this reason, `yquant` offers a compatibility key that is highly recommended to be specified as a package option. This allows certain features that are expected to break old layout or functionality to revert to their previous behavior. Every feature affected by the `compat` key is documented in this manual. Once a compatibility version is selected in a document, it cannot be changed any more. Compatibility versions will include the major and minor, but not the patch level version number (though not every major or minor version will necessarily introduce new compatibility versions). Bugs that clearly violated behavior described in this manual will be fixed without a possibility to revert back. Changes that are not supposed to result in a (more than marginally) different result will not be included in the compatibility layer. If you find this to be wrong in a particular case, please file a bug report.
When starting a new document, it is recommended to leave out the compatibility key at first compilation. `yquant` will then issue a warning from which you can infer the recommended setting, corresponding to the current version. You should then pass the appropriate version to the `\usepackage` command. For example, this manual corresponds to `\usepackage[compat=0.6]{yquant}`.
Allowed values for `compat` are `newest` (discouraged), which equals `0.6`, `0.4`, and `0.3` (default).

2.2 Starred vs. unstarred environment

You may choose to use either the `yquant` or the `yquant*` environment. The former one requires you to define all your registers before you use them (though you may decide to define a register after some operations on *different* registers, but before its first usage).

The starred form additionally supports the use of undeclared registers: it basically declares a registers upon its first usage. This will always be a `qubit` register; but if you use the corresponding attribute and the first usage is an `init` command, you may overwrite this.

Subcircuits always use the unstarred form. New in 0.2

Additionally, if you refer to the index i of a vector register of length $L < i$, this register will automatically be enlarged to $i := L$. It is also possible to convert a scalar register into a vector register in this manner. To enlarge a register in the unstarred environment, you must precede the number of registers to be added in the second declaration by a plus sign. Note that in this manner, you may even create discontiguous vectors. New in 0.1.1

This might be a good point to proceed to the examples section 6.

2.3 Formal syntax

Enhanced in 0.1.2, 0.1.1

Every `yquant` command has the same structure (described here in EBNF syntax):

```
Command = { Arguments }, ?command?, [ Value ], [ RegisterList ], Controls,
↪  ";";
Arguments = "[", ?pgfkeys?, "]";
Value = "{", ?TeX code?, "}";
Controls = [ "|", [ RegisterSingleList ] ], [ "~", [ RegisterSingleList ] ];

RegisterList = (RegisterSingle | RegisterMulti), [ ",", RegisterList ];
RegisterSingleList = RegisterSingle, [ ",", RegisterSingleList ];

RegisterSingle = RegisterSingleNoRange | RegisterRange;
RegisterSingleNoRange = ?name?, [ "[", IndexMultiList, "]" ];
RegisterMulti = "(", ( RegisterMultiNoRange | [ "*" ], RegisterRange ), ")";
RegisterMultiNoRange = [ "*" ], ?name?, [ "[", IndexSingleMainList, "]" ];
RegisterRange = [ RegisterUnique ], "-", [ RegisterUnique ];
RegisterUnique = ?name?, [ "[", ?number?, "]" ];

IndexMultiList = IndexMulti, [ ",", IndexMultiList ];
IndexSingleList = IndexSingle, [ ",", IndexSingleList ];
IndexSingleMainList = [ "*" ], IndexSingle, [ ",", IndexSingleMainList ];
IndexMulti = IndexSingle | ( "(", IndexSingle, ")" );
IndexSingle = ?number? | ( [ ?number? ], "-", [ ?number? ] );
```

Note that `yquant` is quite tolerant with respect to whitespaces. Virtually every comma in the EBNF notation may consist of an arbitrary (including zero) number of whitespaces. Not all combinations that can be constructed by this grammar are actually allowed semantically; but it would make the grammar too verbose to spell this out in detail. Deviations are noted in this manual.

Valid values for ?command? (case-insensitive) are documented in a section 5. We use ?pgfkeys? to describe any valid content passed to the \pgfkeys macro (rather, \yquantset is invoked with some subtleties); and by ?name? we denote any valid register name. Register names must not contain any of the control literals used before (semicolon, comma, parentheses, square brackets, dash, pipe, tilde, beginning star); and you should avoid using special TEX characters. Note that for performance reasons, yquant does not check whether a register name is valid or not, but expect to either see unintended output or not-so-helpful error messages if you choose an invalid name. ?number? is a decimal integer larger or equal to zero (in the context of register creation, strictly larger; in this context, it may also contain a leading "+").

2.4 Registers

Every quantum circuit is structured by means of *registers*. A register has a *type* that specifies how its wire is drawn, and that may even change during its lifetime. At the moment, yquant supports four types:

1. qubit is the most common type, used for a quantum register. It corresponds to a single line.

2. cbit is a classical register, which can be either declared from the beginning or arises by using measurements. It corresponds to a double line.

3. qubits is a "quantum bundle," i.e., a bunch of quantum registers that are always addressed in a group as a single register. Operations between bundles of the same length should be interpreted as transversal. It corresponds to a triple line. An alternative (and more common) representation is to use the qubit type and a slash gate at its very beginning.

4. nobit is the most obscure type, corresponding to a non-existing wire. Mostly, this register type arises by using the discard command. However, it can also be directly declared, which on rare occasions might be necessary (its type can then be changed by means of an init or setwire pseudo-gate). If you want to declare a register only at a certain horizontal position in the circuit, consider using the after argument instead.

Registers must be declared before they can be used (though in the yquant* environment, this declaration may be implicit, creating a qubit register).

Registers can have a vector character, i.e., not only a *name*, but also an *index* (or, in the declaration, a *length*). The index (zero-based) or length is specified

in square brackets following the name, which closely mimics the OpenQASM language.

Vector registers may be non-contiguous: Whenever you create a bunch of registers, it is put at the bottom of the circuit. If you later on again create registers of the same name—either implicitly in the `yquant*` environment, or explicitly by preceding the length of the vectors entries to be added by a plus, as in `qubit a[+3];`—they will be put to what is *now* the bottom of the circuit, even if some other registers are interspersed. New in 0.1.1

Registers are referenced—i.e., used in operations—by their name and index. If the latter is omitted, all indices of the register are targeted. Multiple registers can be referenced by joining their names in a comma-separated list, or by means of a range specifier: give the name of the first (topmost), a dash, and the last (bottom-most) register. Both are inclusive. In a range specifier, omitting the start name means that the range begins at the first known register; omitting the end name means that the range ends at the last known (at the moment of its use) register. Omitting both indicates a range over all known registers.

It is also possible to use comma-separated lists and ranges within the indices themselves, so that, e.g., `a[0, 2, 5-], b[-2]` will target the zeroth and second index of `a`; the remaining indices of `a` starting from five; and the first three indices of `b`. However, if you use an *outer* range (i.e., a range between indices of registers with different names), the initial and final register of the range must be unique, i.e., either you omit the index (targeting the first or last register with the given name) or specify a single one. New in 0.1.1

Ranges and discontiguous registers

Assume a configuration in which the vector register `a` begins with one qubit, then the single register `b` follows, and after that `a` is continued with another qubit.
The range `a-b` will target `a[0]` and `b[0]`, but not `a[1]`. As `a` is used as the initial register in the range without an explicit index specification, `yquant` automatically translates this into `a[0]`, while `b`, being used as the final register, is automatically translated into the last register of name `b` (which here happens to be `b[0]`). Ranges between different register names (outer ranges) are *visual* ranges, i.e., they refer to the top-to-bottom order that is visible. Consequently, the register `a[1]` is left out since it is visually below the others.
Likewise, the range `b-a` will target `b[0]` and `a[1]`.
Ranges within indices are *logical* ranges. Hence, `a`, `a[-]`, `a[0-]`, `a[-1]`, and `a[0-1]` are all equivalent: they all refer to the registers `a[0]` and `a[1]`, but never to `b`, regardless of any visual position.

All that was said so far refers to the operation being carried out on each of the registers *individually*, i.e., producing several copies of the operation. This is different from using the operation multiple times on the individual single registers only with regard to the horizontal positioning: if specified as a register list with one operation, all copies of the operation will be aligned at the same horizontal middle axis (for gates with the same width on each register, this is the same as issuing an `align` command before performing the operations individually).

It is forbidden (in the sense of "not useful and giving unexpected output," but `yquant` does not check for this) to list the same register multiple times (explicitly or via ranges) in one operation.

Instead of copies of single-register operations, one might want to carry out a multi-register operation. In this case, the desired list of registers (comma separated, range, or both) must be surrounded by parentheses. It is possible to mix single- and multi-register operations arbitrarily. In an index list, you may also choose to surround only certain indices with parenthesis, provided the whole register is not already a multi-register.

Note that some gates, such as the `swap` gate, always require (semantically, not grammatically) multi-register operations. The number of constituents is not stipulated; while a `swap` gate with more than two targets is no longer well-defined, other registers such as `zz` may still be useful. `yquant` will prevent you from using a gate in a multi-qubit setting when it may only be used for single registers.

Changed in 0.1.2

Typically, multi-register operations should only be carried out on adjacent registers—but sometimes, one might want to carry out a multi-qubit operation on a visually discontiguous set of registers (which, due to a particular quantum computer topology, might even be physically feasible). `yquant` supports these discontiguous operations explicitly. It will draw a *main* part of the gate at the first contiguous slice of registers in the target list—you may select another register for this part by preceding the name or index with a star (which, contrary to the simplified grammar, may only occur *once* in a target specification). All other contiguous slices of target registers will be drawn in a *subordinate style* for this gate. Finally, all slices will be connected by a single vertical line with the style `/yquant/every multi line`. Subcircuits will always span the full region from the first to the last register specified in a multi-qubit gate. This is due to the fact that they may contain arbitrary ancilla registers which may be positioned somewhere in between the parts that actually constitute the subcircuit—surrounding this with a scattered set of connected boxes would look quite unpleasant.

New in 0.1.2

New in 0.2

Discontiguous targets and control lines

Improved in 0.1.2

A control line extends from the very first to the very last affected register in an operation. A sub-gate line that is used for discontiguous registers will only span the range of a multi-register. This distinction becomes crucial if you want to carry out a *controlled* operation on more than one multi-register, where at least one is discontiguous. Without the controls, the separate multi-registers could be identified, since no connecting vertical line extends between them (unless, which you should strictly avoid, they are intertwined). However, with the controls, the control line will make it hard (for some gates, impossible) to visually distinguish the connected parts. `yquant` will kindly provide a warning in this case. You may choose to suppress this warning using the boolean key `/yquant/operator/multi warning`.

There is no established style for discontiguous gates. Note that at the moment, main and subordinate style coincide for all gates except for the `measure` gate with a value. In order to still make it possible to visually distinguish discontiguous multi-register gates operating on slices of a single register from just a bunch of single-register gates that are executed in a parallel manner if controls are present, `yquant`'s default vertical line style for the former case is a wavy line instead of a straight one. Still, the meaning of this should probably be explained. Please feel free to submit issues or pull requests with propositions of how default styles or alternative subordinate gate shapes may additionally help to mitigate the problem.

2.5 Arguments

Every command may take one or multiple arguments. Those are specified in square brackets that precede the command itself. The content of those square brackets is essentially fed to a `\pgfkeys`-like macro. The default path is set appropriately such that the arguments of the command can be accessed without path specifiers. If the key is not a valid argument for the command or a global argument and it is not given by an absolute path, it is searched for in the `/yquant` namespace. If it cannot be found there, it is passed to `/yquant/operator style`.

Note that commands may have required arguments. If a required argument is missing, an error will be issued.

The `value` attribute can alternatively be given inside curly brackets after the command name and before the register specification. This has the advantage that special characters such as a closing square bracket need not be escaped. If both

alternatives are present, the value inside curly brackets takes precedence and a warning is issued.

2.6 Controls

Lots of gates may have controls, i.e., they are only to be executed if some other gate is set or unset. The former case is called a *positive control*, the latter one a *negative control*. Those are indicated by filled and empty circles on the control registers and a vertical line that joins the registers that belong together.

The gate specification is followed by the list of target registers. By then writing a pipe ("|"), the list of positive controls is introduced; this mimics the mathematical syntax "conditioned on" for probabilities or "given" for sets. If there are no positive controls, the list may be empty or, together with the pipe, omitted. Preceded by a tilde ("~"), the list of negative controls then follows; this mimics the syntax of many programming languages that denote logical negation by a tilde. If there are no negative controls, the list may be empty or, together with the tilde, omitted.

2.7 Importing circuits from files

New in 0.2

yquant provides a simple way to import circuits that are stored in external files. The macro `\yquantimport` can be used in three different contexts:

- Outside of a TikZ picture environment.
  Here, `\yquantimport[<options>]{<filename>}` will be equivalent to

  ```
  \begin{tikzpicture}
     \begin{yquant}[<options>]
        % the content of <filename> goes here
     \end{yquant}
  \end{tikzpicture}
  ```

  The starred form, `\yquantimport*[<options>]{<filename>}`, instead inserts the starred `yquant` environment. Note that the options are always `yquant` options; if you want to pass TikZ options, you will have to create the picture environment by yourself or change the option path to the correct one (`/tikz/.cd`).

- Inside a TikZ picture environment, but outside of a `yquant` environment.
  This is the same as before, just that no extra picture environment will be added.

- Inside both a TikZ picture environment and a yquant environment.
  The file will be inserted directly into the environment. yquant's parser is automatically restarted after this. The content will always be put in a TEX group; if additional options are provided, yquant also inserts a TikZ scope and executes \yquantset{<options>} directly after the scope. If \yquantimport is used, the content will be read as if the containing environment was an unstarred one; if \yquantimport* is used, the content will be read as if the containing environment was a starred one.

Note that yquant internally uses plain TEX's \import command (i.e., \@@import in LATEX). However, when the import package is loaded, it uses \subimport{\yquantimportpath}{<filename>}, where \yquantimportpath defaults to ./—so by changing this, files from other folders may be imported which by themselves again include other files, and the relative path resolution will work.

Note that you may in particular import the content of a subcircuit.

2.8 Defining own gates

Scope

All gate declarations are always global.

If you want to define a gate that corresponds to a single box gate with a certain pre-defined content, you may use the macro \yquantdefinebox{<name>}[<style>]{<content>}, which is far more efficient than the much more general \yquantdefinegate introduced below. It only allows for single-register usage; use \yquantdefinemultibox with the same arguments if you want to allow the gate to be used in a multi-register gate fashion. The macros work in the following way: New in 0.2.1 New in 0.4

- They create a new gate with name <name> that can be accessed as all the other build-in gates. Note that <name> is case-insensitive and may not contain spaces. Special characters are allowed if TEX can cope with them (i.e., no comment signs, no unbalanced braces, no backslashes...).
- They create a style /yquant/operators/every <name> and assign the optional <style> to it. If no style is provided, the default style will inherit from /yquant/operators/every rectangular box. If a compat version before 0.6 is chosen, /yquant/operators/every box will instead be the ancestor. Changed in 0.6

- They define <content> to be the value that is written into the box. This <content> is expanded in a protected manner at the time of gate declaration. You may need to prefix fragile macros by \protect.

Sometimes, you may wish to define gates that are more than just a single box— perhaps a succession of multiple gates or even multi-register gates with individual operations on the input registers. yquant provides a simple macro that allows this. The macro \yquantdefinegate{<name>}[<style>]{<content>} works in the following way: New in 0.2

- It creates a new gate with name <name> that can be accessed as all the other built-in gates. Note that <name> is case-insensitive and may not contain spaces. Special characters are allowed if TEX can cope with them (i.e., no comment signs, no unbalanced braces, no backslashes...).
- It creates a style /yquant/operators/every <name> and assigns the optional <style> to it. If no style is provided, the default style will inherit from /yquant/operators/every custom gate. This will make the gate "seamless," i.e., avoid highlighting the fact that this is a custom gate.
- It defines a macro that contains <content> (expanded in a protected manner) and that will be inserted as a subcircuit whenever this gate is invoked. This in particular means that if you use \yquantimport within the gate, the file will only be loaded once at the time of declaration.

When the gate is later drawn, the styles are invoked in the following order— remember custom gates are implemented by means of subcircuits—:

1. /yquant/every operator
2. /yquant/operators/every <name>
3. /yquant/operators/every subcircuit box
4. /yquant/this operator
5. /yquant/operators/this subcircuit box

Gates defined in this way can only make use of the default gates or other custom gates. They do not accept custom arguments, and it is not possible to declare own, custom shapes in this way (though other predefined shapes may be used). If they are used in a multi-qubit manner, they will never be split into contiguous slices (but their content will be, so if you use the default style that turns off the box, the only way to notice this is that intermediate unaffected registers will not be allowed to have gates visually within the rectangle that would bound the custom gate).

Redefining existing gates

The above macros will issue an error if the gate already exists. You can use `\yquantredefinebox`, `\yquantredefinemultibox` (use the appropriate command for the *new* definition), or `\yquantredefinegate` to overwrite existing gate definitions. Note that this will overwrite *any* gate, even the built-in ones. Generally, it is discouraged to make use of this possibility. For custom gates, if you redefine a gate as a box which was previously a general subcircuit-based gate, the macro that contains the subcircuit will still be held in memory. Overwriting built-in gates will not clear the attributes associated to this gate (though required attributes will no longer be required afterwards). Again, this is not a problem but prevents `yquant` from issuing potentially helpful error message if such a—now meaningless—attribute is used.
Finally, once a built-in gate is overwritten, it cannot be restored. In particular, the register creation pseudo-gates `qubit`, `cbit`, `qubits`, and `nobit` perform some magic that cannot be mimicked with custom gates.

More advanced declaration of custom gates requires the use of backend macros. Refer to `yquant-lang.tex` for this. Note that the backend interface changed in version 0.4. For the declaration of custom shapes, see `yquant-shapes.tex` for examples.

2.9 The flow of time: horizontal and vertical layout

New in 0.7

By default, quantum circuits are oriented *horizontally*, i.e., time flows from left to right. Sometimes, this is problematic when printed on a portrait page layout and a *vertical* layout would be better suited. `yquant` supports this by simply passing the configuration option `/yquant/vertical` to the circuit (this can even be done globally, so that all circuits are rendered vertically).

The vertical layout is supposed to work exactly as the horizontal layout; however, most testing occurs for horizontal mode only (basically, for every release, the examples in this manual comprise the testsuite). Hence, if something weird happens in vertical mode, please file a bug report.

Most of the internal and exposed nomenclature in `yquant` is designed for the horizontal case (names such as “height” or “width”). For some of the exposed configuration options, synonyms are provided which are more meaningful in vertical mode or that are more orientation-agnostic. These synonyms are merely conveniences, they do not provide any new functionality. As a general rule of thumb, everything that was “atop” becomes “left.”

Some relevant TikZ options refer to a fixed direction, namely `[x|y] radius`

and `[inner|outer] [x|y]sep`. If you think that maybe you want to change the orientation of your circuit at some point, you should never use the original TikZ styles—their meaning would depend on the current orientation. Instead, consider using the `yquant` alternatives `[time|space] radius` and `[inner|outer] [time|space]sep`.

3 Configuration

yquant uses pgfkeys to control its options, which are located in the path /yquant. The following list contains all options and styles that are recognized, apart from gate arguments. Those are listed together with their operations.

3.1 Circuit layout

/yquant/register/minimum height default: 1.5mm Changed in 0.7, 0.4

yquant automatically determines the height (extent from wire to top boundary; in vertical mode: from wire to left boundary) of a register as the height of the largest operation. This might be too small for two reasons:

- if the register is used only with small gates (e.g., only as a control, or as a swap), and it does not have a label (or one containing only x-height letters).
- if you manually turned off height calculation or multi-extent calculation for a large gate. yquant will then not consider the vertical extent of this gate, which might consequently lead to undesirable overlaps.

This key provides an easy alleviation of the problem by requiring a minimal height for every register. As the value of this key is relevant at the time of register declaration, it can also be changed for each register individually.
Note that in vertical mode, the default of this setting is 2.5mm.
Note that this key is affected by the compat setting. Before version 0.4, there was no /yquant/register/minimum depth key. In this compatibility setting, passing the value x to this key will set both height and depth to $\frac{x}{2}$. The default for x is then 3mm.

/yquant/register/minimum depth default: 1.5mm Changed in 0.4 New in 0.4

see /yquant/register/minimum height

This key allows to specify a minimum depth (extent from wire to bottom boundary; in vertical mode: from wire to right boundary) for a register.
Note that in vertical mode, the default of this setting is 2.5mm.
Note that this key is affected by the compat setting. Before version 0.4, this key will not be available.

/yquant/register/minimum left default: 1.5mm New in 0.7

This is a synonym for /yquant/register/minimum height, as this naming makes more sense for vertical circuits.

`/yquant/register/minimum right` default: `1.5mm` New in 0.7

This is a synonym for `/yquant/register/minimum depth`, as this naming makes more sense for vertical circuits.

`/yquant/register/minimum before` default: `1.5mm` New in 0.7

This is a synonym for `/yquant/register/minimum height`, which provides a naming that makes sense in both horizontal and vertical mode.

`/yquant/register/minimum after` default: `1.5mm` New in 0.7

This is a synonym for `/yquant/register/minimum depth`, which provides a naming that makes sense in both horizontal and vertical mode.

`/yquant/register/separation` default: `1mm`

This key controls the amount of vertical space that is inserted between two successive registers. Half of this value is also the length that multi-`init` or multi-`output` braces extend beyond the mid position of the register.

`/yquant/operator/minimum width` default: `5mm` Changed in 0.7

`yquant` automatically determines the width (in vertical mode: the vertical extent) of an operator according to its content. However, single-letter boxes are among the most common operators, and giving them slightly different widths would result in a very uneven spacing, as `yquant` does not use a grid layout but stacks the operators horizontally one after each other. Hence, this key provides a minimum width that will be set for every operator. This does not imply that the *visual* appearance (i.e., the `x radius` key) is enlarged, but that operators of a smaller actual width will be centered in a virtual box of the minimum width.
Note that in vertical mode, the default of this setting is `3mm`.

`/yquant/operator/minimum extent` default: `3mm` New in 0.7

This is a synonym for `/yquant/operator/minimum width`, which provides a naming that makes sense in both horizontal and vertical mode.

`/yquant/operator/separation` default: `1mm`

This key controls the amount of horizontal space that is inserted between two successive operators and at the beginning and end of a circuit.

`/yquant/operator/multi warning` default: `true` New in 0.1.2

If this key is `true`, a warning is displayed whenever more than a single multi-register gate, where at least one is discontiguous, is employed together with controls. Even if a visual distinction between control and multi-qubit line may be possible (depending on the style in use), they will overlap and produce unaesthetic output. You may disable this warning globally, on a per-circuit, or even on a per-gate basis.

`/yquant/drawing mode` default: `quality` New in 0.7

This key determines which drawing pipeline is enabled. For more details, see section 4.3. The option should not be changed within a circuit (though only the last value will be relevant). Allowed values are `quality` for the default clipping-based pipeline, and `size` of the filling-based one. Note that choosing `quality` will set `/yquant/default background` to `none`; choosing `size` will set it to `white` (though this can be changed after setting the option).

`/yquant/default background` default: `none` New in 0.7

This key contains the default color that is used to fill all gates with a nonempty interior.

`/yquant/default fill` default: `fill/.expanded=\pgfkeysvalueof{/yquant/default background}` New in 0.7

Use this style if you want to apply the default filling to a user-defined gate.

3.2 Register creation

`/yquant/register/default name` default: `\regidx`

The printed name that is used by default if a new register is created explicitly (`qubit`, `cbit`, `qubits`; not used for `nobit` or for implicit declarations) and no value is specified. The following macros are available:

- `\reg` contains your name to identify this register.
- `\idx` contains the index (zero-based) of the current register within a vector register.
- `\regidx` expands to `\reg` if the register is of length one, and to `\reg[\idx]` else.
- `\len` contains the length of the current register vector.

/yquant/register/default lazy name default: New in 0.6

The printed name that is used by default if a new register is created implicitly (i.e., without using any of qubit, cbit, qubits, or nobit, but inside a yquant* environment by just using the register). The same macros as with /yquant/register/default name are available. Note that this default setting is not used when the register is created via an init gate—its value always overwrites the default.

/yquant/every label default: shape=yquant-init, anchor=center, Changed in 0.7, 0.4
align/.expanded=\ifyquanthorz{right}{center}, outer timesep=2pt,
/yquant/operator/if multi={draw, decoration={gapped brace,
raise=2pt, \ifyquanthorz{mirror}{}}, decorate}

This style is installed for every single register name label (i.e., upon creation and when used with the init gate). The default style allows to use line breaks in the labels. The node shape, yquant-init, will generate a path at its right side (in vertical mode: at its bottom side), which is replaced by the gapped brace decoration if the gate is used in a multi-register fashion. The decoration is similar to TikZ's brace decoration, but additionally allows specify the regions in which a line should be drawn by using the /tikz/decoration/from to key, which expects a comma-separated list of dimension ranges, and which is automatically populated by yquant.

Note that if the compat key is below 0.3, the multi options are instead read from /yquant/every multi label.

/yquant/every initial label default: Changed in 0.7
anchor/.expanded=\ifyquanthorz{east}{south},
/yquant/internal/autorotate init

This style is installed for every single register name label at the left border of the circuit. It is therefore used if a label is specified upon declaration and also for the init gate if it happens to be the first gate on an unlabelled register (use a zero-width hspace gate before if you want to suppress this). Changed in 0.4

The automatic rotation will be set up by using the /yquant/vertical style with an argument; by default, it is empty. New in 0.7

/yquant/every qubit label default:

This style is installed for every single register name label of a register of type qubit.

/yquant/every cbit label default:

This style is installed for every single register name label of a register of type cbit.

`/yquant/every qubits label` default:

This style is installed for every single register name label of a register of type `qubits`.

`/yquant/every multi label` default: `draw, decoration={gapped brace, mirror, raise=2pt}, decorate` Removed in 0.4 Changed in 0.1.2

This style is installed for every register name label that is attached to a multi-qubit register by means of the `init` gate.

Note that this key is only available if the `compat` setting is smaller than `0.4`. In newer versions, this is incorporated in `/yquant/every label`.

`/yquant/every input label` default: Removed in 0.4 New in 0.2

This style is installed for every register name label in a `subcircuit` when the register is an input (or input and output) register.

Note that this key is only available if the `compat` setting is smaller than `0.4`; and in this case, it behaves inconsistently, as it is only applied for labels directly specified during creation, but not for initial `init` gates.

3.3 Register outputs

`/yquant/every output` default: `shape=yquant-output, anchor/.expanded=\ifyquanthorz{west}{north}, align/.expanded=\ifyquanthorz{left}{center}, outer timesep=2pt, /yquant/operator/if multi={draw, decoration={gapped brace, raise=2pt, \ifyquanthorz{}{mirror}}, decorate}, /yquant/internal/autorotate output` Changed in 0.7, 0.4

This style is installed for every `output` label at the end of the circuit. The default style allows to use line breaks in the labels.

The node shape, `yquant-output`, will generate a path at its left side (in vertical mode: at its top side), which is replaced by the `gapped brace` decoration in the case of multi-register usage. See `/yquant/every label` for a more detailed explanation.

The automatic rotation will be set up by using the `/yquant/vertical` style with an argument; by default, it is empty. New in 0.7

`/yquant/every qubit output` default:

This style is installed for every `output` label of a register of type `qubit`.

`/yquant/every cbit output` default:

This style is installed for every `output` label of a register of type `cbit`.

`/yquant/every qubits output` default:
This style is installed for every `output` label of a register of type `qubits`.

`/yquant/every multi output` default: `draw, decoration={gapped brace, raise=2pt}, decorate` Removed in 0.4 Changed in 0.1.2
This style is installed for every `output` label that is attached to a multi-qubit register.
Note that this key is only available if the `compat` setting is smaller than `0.4`. In newer versions, this is incorporated into `/yquant/every output`.

3.4 General styling

`/yquant/every circuit` default: `every node/.prefix style={transform shape}` Changed in 0.4, 0.1.2
Style that is installed for every `yquant` and `yquant*` environment, as if it had been given as an option. The style's default path is, as with all other styles, `/tikz`. The style is re-applied for every subcircuit. The default style will make all nodes (which in particular means, all gates) respect outer canvas transformations.
If your TikZ version is before 3.1.6a, this style will additionally contain `every label/.prefix style={transform shape=false}`, which undoes the effect for `labels` (see TikZ bug #843). An update is recommended.

`/yquant/every wire` default: `draw`
This style is installed whenever a wire is drawn.

`/yquant/every qubit wire` default:
This style is installed whenever a wire for a register of type `qubit` is drawn.

`/yquant/every cbit wire` default:
This style is installed whenever a wire for a register of type `cbit` is drawn.

`/yquant/every qubits wire` default:
This style is installed whenever a wire for a register of type `qubits` is drawn.

`/yquant/every control line` default: `draw`
This style is used to draw the vertical control line that connects controlled operations and their controls.

`/yquant/every control` default: `shape=yquant-circle, anchor=center, radius=.5mm`
This style is used to draw the node for a control, both positive and negative.

`/yquant/every positive control` default: `fill=black`

This style is installed for every positive control (i.e., one that conditions on the register being in state $|1\rangle$ or 1).

`/yquant/every negative control` default: `draw, /yquant/default fill` Changed in 0.7

This style is installed for every negative control (i.e., one that conditions on the register being in state $|0\rangle$ or 0).

`/yquant/every operator` default: `anchor=center`

This style is installed for every gate (and also pseudo-gates such as the `slash` operator) that acts on one or multiple registers.

`/yquant/every multi line` default: `draw, decoration={snake, amplitude=.25mm, segment length=5pt}, decorate` New in 0.1.2

This style is used to draw the vertical line that connects discontiguous slices of sub-gates.

`/yquant/this operator` default:

This style is appended to the current style installed for an operator; it should be used only locally to overwrite any global configuration effect.

`/yquant/this control` default:

This style is appended to the current style installed for a control; it should be used only locally to overwrite any global configuration effect.

`/yquant/operator style` default: `/yquant/this operator/.append style={#1}`

This is a shorthand that can be used to modify the appearance of the current operator.

`/yquant/control style` default: `/yquant/every control line/.append style={#1}, /yquant/this control/.append style={#1}`

This is a shorthand that can be used to modify the appearance of the current control and its associated line.

`/yquant/style` default: `/yquant/operator style={#1}, /yquant/control style={#1}`

This is a shorthand that modifies the appearance of both the current operator and any controls or control lines.

/yquant/operator/multi as single default: /yquant/every multi line/.style=/yquant/every control line Changed in 0.7.1 New in 0.1.2

This style is automatically set for certain gates such as the swap or the zz gate. For those gates, neighboring registers will be treated as discontiguous; and this style will enforce their connecting line to have the style used by control lines.

This style actually checks whether control lines are present in the gate and in this case, it is equal to draw=none, i.e., it suppresses the multi-register line. This is due to the fact that this line would be drawn on top of (a segment of) the control line with an identical style, so that it cannot be seen unless there were some bug in the renderer. Note that the check for control lines is embedded into the style (the default value shown here is a simplification); so if you change it, you will overwrite the style for *all* cases, not only when there is no control line. New in 0.7.1

The default /yquant/every multi line is a wavy line; this allows to distinguish discontiguous multi-qubit gates from multiple single-qubit gates when using controls. Still, some gates have such an established appearance that—despite being logically misleading—we rather use the same style as for a control line.

/yquant/operator/if multi default: New in 0.4

This style can be invoked by other styles with an arguments that contains styles to be executed only if the current gate is used in a multi-register fashion. See /yquant/every label for an example.

/yquant/circuit/seamless default: false New in 0.4

The value of this setting determines whether circuits drawn in a yquant environment in the current group will be drawn in a “seamless” state (hence, this style must be set *before* the yquant environment is started). The key /yquant/operator/separation will control the amount of padding with which a wire starts or ends before the first or after the last gate. By turning on the seamless state, this padding is suppressed. Using outputs or giving an initial value at the register declaration brings the corresponding padding back. Usually, this setting is intended only for subcircuits. Direct access is discouraged, as it will persist in subcircuits. Only access it via /yquant/operators/subcircuit/seamless.

/yquant/circuit/orientation default: horizontal New in 0.7

This setting allows two possible values, horizontal and vertical. It must only be changed before a circuit or at latest with the option arguments to the yquant environment, but not within a circuit.
In the default horizontal mode, time flows from left to right and registers will be created from top to bottom. In the alternative vertical mode, time flows from top to bottom and registers will be created from left to right.
Note that this setting influences the behavior of various macros and styles:

Macro/Style	horizontal meaning	vertical meaning
\ifyquanthorz{a}{b}	a	b
/tikz/time radius	/tikz/x radius	/tikz/y radius
/tikz/space radius	/tikz/y radius	/tikz/x radius
/tikz/inner timesep	/tikz/inner xsep	/tikz/inner ysep
/tikz/inner spacesep	/tikz/inner ysep	/tikz/inner xsep
/tikz/outer timesep	/tikz/outer xsep	/tikz/outer ysep
/tikz/outer spacesep	/tikz/outer ysep	/tikz/outer xsep

Additionally, this setting influences the default values of various yquant styles—note that if a style was once overwritten, the user-supplied value will be never be changed! Here, we use the orientation-independent names of the styles, although all their synonyms are equivalent.

Style	horizontal def.	vertical def.
/yquant/register/minimum before	1.5mm	2.5mm
/yquant/register/minimum after	1.5mm	2.5mm
/yquant/operator/minimum extent	5mm	3mm

/yquant/horizontal default: New in 0.7

This style is a shorthand that sets /yquant/circuit/orientation to the horizontal value (which is the default).

`/yquant/vertical` default: 0 New in 0.7

This style is a shorthand that sets `/yquant/circuit/orientation` to the `vertical` value. Additionally, it accepts a parameter which should be an angle value (in degrees) between `-180` and `180`. This is a rotation that is automatically applied to the text in every `init` gate at the beginning of a circuit; the inverse rotation is automatically applied to the text in every `output` gate at the end of a circuit.

Note that the gates themselves are *not* rotated. This would be counterproductive for multi-qubit gates that contain braces—those should still be orthogonal to the wire lines. In order to achieve this, the macro will use internal TikZ details and exploit that the `execute at begin node` key is directly followed by the braced content of the node. Hence, it will add a corresponding `\adjustbox{rotate=#1}` to this style. Therefore, `yquant` will raise a warning if the argument is used and the `adjustbox` package is not loaded, in which case the rotation is just ignored.

`/yquant/every post measurement control` default: `indirect` New in 0.4

This style determines the default arrangement of measurements that are followed by positive controls.

The default option `indirect` will draw the measurement at the position where it is specified. Any later use of a control will be at the position of the controlled gate.

The option `direct` will defer the measurement. If later on, a controlled operation is used where the positive controls contain all of the targets of this measurement *and* no other gate was executed meanwhile on any of the targets of this measurement, then the measurement gate will replace the corresponding positive control knobs (and might inherit TikZ options of the embedding gate); otherwise, it will behave as if the `indirect` option had been specified.

Some care must be taken when gates are named that are affected by this option. If the embedding gate is named, the positive controls that will be replaced by measurements are no longer available with the "p" suffix (but other positive controls will still be numbered as if all were). Attach the name to the measurement in order to access it as if it were an ordinary gate; however, note that the name only becomes available after the later embedding gate was called.

Note that this setting affects all measurements that have a compatible shape; currently, this is only `measure`. While there is no technical difficulty in implementing the same behavior for `dmeter`, its particular shape does not really suggest this use. However, if you desire to do so, please file a feature request.

3.5 Styles for operators

`/yquant/operators/every barrier` default: `shape=yquant-line, dashed, draw, shorten <= -1mm, shorten >= -1mm` — Changed in 0.4

This style is installed for every `barrier` pseudo-gate, i.e., the one that is used to explicitly denote a separation between "before" and "after" within the circuit. Note that the `shorten` keys are only present in the default style if you specify at least the compatibility version `0.4`.

`/yquant/operators/every box` default: `/yquant/operators/every rectangular box` — Changed in 0.6

This style is installed for every `box` operator. Note that with a `compat` setting strictly smaller than `0.6`, the definition of this style was the one that is now `/yquant/operators/every rectangular box`, and this style was also the base style from which all box-like gates inherited. With a `compat` setting of at least `0.6`, no other gates apart from `box` will use this style directly or indirectly.

`/yquant/operators/every custom gate` default: `/yquant/operators/subcircuit/seamless` — Changed in 0.4, New in 0.2

This style is by default installed for every user-defined gate. User-defined gates are implemented via subcircuits; this style suppresses the box that surrounds the subcircuit and by default suppresses all register names. This allows a seamless integration of the gate/subcircuit into the main circuit, without putting particular emphasis to the fact that what was defined as the custom gate indeed belongs together. Note that with the `compat` key set before `0.4`, this style instead defaults to `/yquant/operators/subcircuit/frameless, /yquant/register/default name=`.

`/yquant/operators/every dmeter` default: `shape=yquant-dmeter, time radius=2mm, space radius=2mm, draw, /yquant/default fill` — Changed in 0.7

This style is installed for every `dmeter` gate. The `yquant-dmeter` shape consists of a rectangle whose right side is replaced by a circle, resembling the letter "D."

`/yquant/operators/every h` default: `/yquant/operators/every rectangular box`

This style is installed for every `h` (Hadamard) operator.

`/yquant/operators/every inspect` default: `shape=yquant-output, align/.expanded=\ifyquanthorz{left}{center}, outer timesep=.3333em, space radius=2.47mm, /yquant/default fill, /yquant/operator/if multi={draw, decoration={gapped brace, raise=2pt, \ifyquanthorz{}{mirror}}, decorate}` Changed in 0.7

This style is installed for every `inspect` gate. It does not have any shape on its own, apart from multi-register uses, in which it will contain a brace on its left (in vertical mode: on its top).

`/yquant/operators/every iswap` default: `shape=yquant-ocross, radius=.75mm, draw` New in 0.7.2

This style is installed for every `iswap` gate that interchanges two qubits and conditionally adds phases. The `yquant-iswap` shape is a cross enclosed in a circle.

`/yquant/operators/every measure` default: `shape=yquant-measure, x radius=4mm, y radius=2.5mm, draw, /yquant/default fill` Changed in 0.7

This style is installed for every `measure` gate. The `yquant-measure` shape is a rectangle that contains a "meter" symbol. It allows for a text to be put inside (e.g., a basis), which then shifts the meter symbol accordingly.

`/yquant/operators/every measure meter` default: `draw, -{Latex[length=2.5pt]}`

This style is applied to the path that resembles the "meter" symbol that is drawn by the `yquant-measure` shape. Due to the default style, the TikZ library `arrows.meta` is automatically loaded with `yquant`.

`/yquant/operators/every not` default: `shape=yquant-oplus, radius=1.3mm, draw`

This style is installed for every `not` or `cnot` gate (which are synonyms, and actually do the same as the Pauli σ_x gate). The `yquant-oplus` shape resembles the addition-modulo-two symbol $\oplus$.

`/yquant/operators/every pauli` default: `/yquant/operators/every rectangular box`

This style is installed for every Pauli operator, i.e., `x`, `y`, and `z`.

`/yquant/operators/every phase` default: `shape=yquant-circle, radius=.5mm, fill`

This style is installed for every `phase` gate $|0\rangle\langle 0| + \mathrm{e}^{\mathrm{i}\phi}\,|1\rangle\langle 1|$.

`/yquant/operators/every rectangular box` default: `shape=yquant-rectangle, draw, align=center, inner timesep=1mm, time radius=2mm, space radius=2.47mm, /yquant/default fill` Changed in 0.7 New in 0.6

This style is not associated to any particular gate, but will be inherited by a lot of gates that have a rectangular box frame with some text. This style should not be used with a `compat` setting strictly smaller than `0.6`.

`/yquant/operators/every slash` default: `shape=yquant-slash, x radius=.5mm, y radius=.7mm, draw`

This style is installed for every `slash` pseudo-gate, i.e., the one that is used to indicate that a single register line actually denotes multiple registers.

`/yquant/operators/every subcircuit` default: New in 0.2

This style is installed for every `subcircuit`. Note that all styles given here will also apply to every element in the subcircuit; in a way, this is an addition to `/yquant/every circuit` (which is also again put into effect at the beginning of a subcircuit).

`/yquant/operators/every subcircuit box` default: `/yquant/operators/every rectangular box, fill=none` New in 0.2

This style is installed for every `subcircuit`. Note that in contrast to all other styles such as `/yquant/operators/every subcircuit` or `/yquant/this operator`, this style is only applied to the "container" node of the subcircuit, but not to the elements in the subcircuit themselves. Also note that the box style by default contains an `inner xsep` that will be added as an inside padding. This makes sense if your wires have labels so that these labels don't move too closely to the border of the box. However, if you do not labelled wires but still want to have a box around the subcircuit, you should consider removing the separation—as it will be added to the initial wire padding given by `/yquant/operator/separation`.

`/yquant/subcircuit box style` default: `/yquant/operators/every subcircuit box/.append style={#1}` New in 0.2

This is a shorthand to append styles to the subcircuit box only.

`/yquant/operators/this subcircuit box` default: New in 0.2

This style is appended to the current style installed for the `subcircuit`, but will not apply to its contents. Additionally, this style will be reset to an empty style at the beginning of each subcircuit, so that it really only applies to exactly the subcircuit box it is explicitly specified on, not to nested subcircuit boxes.

/yquant/this subcircuit box style default: /yquant/operators/this subcircuit box/.append style={#1} New in 0.2

This is a shorthand to append styles to the current subcircuit box only.

/yquant/operators/subcircuit/frameless default: /yquant/operators/this subcircuit box/.append style={draw=none, inner sep=0pt} New in 0.4

This is a shorthand style that removes the frame and additional inner separation for the current subcircuit. Note that still, the wire padding given by /yquant/operator/separation is present within the—now invisible—outer box that contains the subcircuit (use /yquant/operators/subcircuit/seamless to suppress it). Hence, the most prominent application of this key is if the wires before and after the subcircuit are nobits, which provides a clean way to build up circuit equations with perfectly aligned wires (examples can be found in section 6).

`/yquant/operators/subcircuit/name mangling` default: `prefix or discard` New in 0.5

This option defines how named nodes within subcircuits are made available to the outer circuit:

- `prefix or discard`
  If the subcircuit itself has a name s and the inner gate has a name g, the outer circuit can refer to the inner gate via the name s-g. Note that s itself may not only consist of the user-specified name, but may instead already be suffixed by `-0`, `-1`, …, if the subcircuits were assigned to multiple targets.

  If the subcircuit itself has no name, works as `discard`.

- `prefix or transparent`
  If the subcircuit itself has a name, as works as `prefix or discard`; else, works as `transparent`.

- `transparent`
  The inner gates are always available in the outer circuit by their original names. Note that this may potentially lead to naming conflicts, which are always resolved by the latest name overwriting all previous declarations without notice.

- `discard`
  The inner gates will not be available in the outer circuit.

Note that if a new gate is defined via `\yquantdefinegate`—which internally uses subcircuits—the value of this option at the time of declaration is the relevant one, not the one at the time of usage. This can be influenced via `/yquant/operators/subcircuit/name mangling reset`.

`/yquant/operators/subcircuit/name mangling reset` default: `true` New in 0.5

The current value of `/yquant/operators/subcircuit/name mangling` will be reset at the beginning of a subcircuit to the value it had upon declaration of the subcircuit only if this option is `true` upon *using* the subcircuit (which typically will only make a difference if the subcircuit was defined via `\yquantdefinegate` at some earlier stage).

Setting this value to `false` has the potential of breaking the corresponding subcircuit/custom gate, as it may internally reference gates by names that are no longer the correct ones. Do not use this property without a very good reason and thorough understanding of what is happening.

/yquant/operators/subcircuit/seamless default: /yquant/operators/subcircuit/frameless, /yquant/register/default name=, /yquant/circuit/seamless New in 0.4

This option carries out multiple actions that are responsible to let the current subcircuit appear in a "seamless" state:

- It calls /yquant/operators/subcircuit/frameless.
- It sets /yquant/circuit/seamless to true.
- It ensures that /yquant/circuit/seamless is reset within the subcircuit, so that it does not propagate to nested subcircuits.

/yquant/operators/every swap default: shape=yquant-swap, radius=.75mm, draw

This style is installed for every swap gate that interchanges two qubits. The yquant-swap shape consists of a single cross.

/yquant/operators/every text default: shape=yquant-rectangle, align=center, inner timesep=1mm, time radius=2mm, space radius=2.47mm, /yquant/default fill Changed in 0.7 New in 0.6

This style is installed for every text gate.

/yquant/operators/every wave default: shape=yquant-circle, radius=.5mm, fill New in 0.1.2

This style is installed for every correlate gate.

/yquant/operators/every x default: /yquant/operators/every pauli

This style is installed for every Pauli operator σ_x, i.e., x.

/yquant/operators/every xx default: shape=yquant-rectangle, radius=.75mm, draw, /yquant/default fill Changed in 0.7

This style is installed for every xx gate in symmetrized notation ($|{+}{+}\rangle\langle{+}{+}| + |{+}{-}\rangle\langle{+}{-}| + |{-}{+}\rangle\langle{-}{+}| - |{-}{-}\rangle\langle{-}{-}|$).

/yquant/operators/every y default: /yquant/operators/every pauli

This style is installed for every Pauli operator σ_y, i.e., y.

/yquant/operators/every z default: /yquant/operators/every pauli

This style is installed for every Pauli operator σ_z, i.e., z.

`/yquant/operators/every zz` default: `shape=yquant-circle, radius=.5mm, fill`

This style is installed for every `zz` gate (aka CPHASE) in symmetrized notation ($|00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 10| - |11\rangle\langle 11|$).

4 Doing the impossible

yquant will almost certainly never be able to do everything an author has in mind. Sometimes, there is the need to draw something non-standard, and this cannot be implemented in the yquant language. However, since yquant is a layer on top of TikZ, it should be very hard to find something (meaningful) that cannot be done by combining the power of both packages.

4.1 Mixing yquant and TikZ code

Before or after any gate, you may interrupt the yquant instructions to perform arbitrary TikZ path operations. After every such operation, yquant will automatically restart its parser so that you can fluently jump between yquant and TikZ code. You can even interject arbitrary TEX code (or, say, low-level pgf commands); however, then, yquant is not able to restart its parser. For this reason, after the last command in a block of TEX commands, you must issue `\yquant`, which then re-enables the yquant language.

4.2 Accessing gates in TikZ

The feature to perform arbitrary TikZ operations is powerful in itself, but would be of limited use were there no way to access the elements in the quantum circuit. yquant provides a global attribute `name` that can be assigned to every gate. All quantum operations are in fact TikZ nodes, and the name you give to them then becomes a TikZ name, which you can easily reference to get the coordinates of a particular operator. Note that the name you specify is only available if a single register is targeted. The name is suffixed by `-\idx`, where `\idx` refers to the (zero-based) index of the operation ordered from top to bottom (i.e., if an operator acts on two qubits and should be named `op`, the topmost operator will be available as `op-0` and the second as `op-1`). Multiple slices in a discontiguous multi-register are additionally suffixed by `-s<`*slice index*`>`. All controls are also named, suffixed by `-p\idx` or `-n\idx` for positive and negative controls (i.e., the topmost positive control of the previous operator will be available as `op-p0`). Counters for target registers, positive, and negative controls are all independent. Finally, you can even access names within a subcircuit, provided you give a name to the subcircuit. All nodes in the subcircuit will then have the name *<subcircuit name>-<name specified in the subcircuits>*. Note that here, *<subcircuit name>* is the *full* name of the subcircuit, which includes the `-\idx` suffix, unless there is only a single target register. For nested subcircuits, you will get multiple prefixes. The

New in 0.4.1
New in 0.5

prefixing behavior can be influenced by /yquant/operators/subcircuit/name mangling.

4.3 Shapes and the drawing pipeline

All yquant shapes have the anchors available you would typically expect from a TikZ shape of the given outline. The center anchor will be aligned to the wire. In addition to the normal paths implemented by TikZ shapes, the width and height of those fit for yquant at least twice as large as given by the /tikz/x radius and /tikz/y radius; and they must implement clipping paths, a yquant addition to TikZ shapes. Such a path has to provide the "clipping outline," i.e., anything that should not contain register or control lines. There may be a difference between horizontal and vertical clipping outlines. To understand clipping paths, yquant's drawing pipeline needs to be explained.

Changed in 0.1.2

- In a first run—this is what happens directly at the position where you type the gate command—yquant will "virtually" draw the gates in order to determine their dimensions and calculate register heights. The actual drawing commands are written to a macro (this is the cause that some macros must be preceded by \protect if used in a gate value—in fact, if multiple registers are targeted in one gate, the style and values required for this gate are only stored once, so that for example \idx is a \protected macro until the very end).
- Deferred gates (measurements that may replace future control knobs) are stored temporarily and queried when the next gate is executed or at the end of the circuit. The corresponding commands—either re-inserting if they must appear at their original position or substituting the controls—are inserted appropriately.

  New in 0.4
- When \end{yquant} is encountered, the vertical positions are determined and the actual drawing commands are executed.
- Unless the operation changes the wire type or style, do the following (first two items for every register at which an operator node has to be created).
  - Create the operator node at the appropriate position.
  - Call \pgfshapeclippath on the newly created node. This will first determine whether the node was stroked; if not, \pgflinewidth is set to zero. Then, it will call the horizontal clipping path, which is supposed to create some soft path commands. Those soft path commands are collected in a macro on a per-register basis and the soft path is cleared.

The same happens for the vertical clipping path, which is collected in a macro on a per-operation basis.

  - If control lines or multi lines are to be drawn, the vertical clipping path commands are now executed and installed as an inverted[1] clipping.
  - Control lines and multi lines are drawn (in this order) from one to the next `center` anchor. Due to the clipping commands, this will create a perfect connection with the shape of the gate, but even transparent gates are possible without the lines being visible.

- If the operation changes the wire type or style, or if there is no operation left on this register, the following is done.
  - Load the clipping paths accumulated for all the gates acting on this register and install the inverted clipping.
  - Draw the wire as one continuous line from where the last wire ended (or the beginning of the circuit) to the center of the last gate, or to the common end position for all wires of the circuit.
  - Remove the clipping paths stored so far on this register, apart from the clipping on the last gate (which will be needed again if this was not the end of the circuit).

Note that yquant also supports a simplified drawing pipeline which does not involve clipping paths. It can be enabled by setting the `/yquant/drawing mode` option to `size`. The simplified pipeline has the following benefits: New in 0.7

- TEX has to compute a bit less, so the compilation process can be sped up.
- PDF readers may render the circuit faster.
- The size of the output files is decreased.
- Compatibility with very simplistic clients that don't support clipping well is improved (this may be an issue for some PDF to image converters, for example).
- Theoretically, you may use any kind of predefined shape in this mode (from TikZ or other packages), since the clipping paths are no longer required.

[1] Inverting the clipping means that instead of drawing only *within* the clipping path (which corresponds to the gates), we only draw *outside*. However, as there is no direct support for this, we invert by exploiting the even-odd rule. If you specify a register multiple times, whether as target, control, or mixed, funny effects can be expected, as the clipping region is inverted multiple times. Note that using a register more than once is always an error, but yquant does not check for it due to the high overhead.

However, with it come certain drawbacks:

- Gates are no longer transparent, but filled with some fixed background color, so if transparency is important for you, this just does not work.
- `yquant` relies on TikZ's layering capabilities; if you use layers by yourself, you have to pay some attention to do it correctly.
- The z-order may not be as you expect it; in particular: All the gates will be drawn on the `main` layer; all the wires on the `wires` layer, which is behind `main`; so if you don't use layers by yourself, everything that you draw will overshadow the wires, but not necessarily the gates.
- If you were to fill a subcircuit, this would then erase all the internal wires. Hence, the box of a subcircuit is drawn on the `behindwires` layer; but this implies that also the *frame* of the box is drawn behind the wires and that this box is also overshadowed by any of your drawing.
- Wires that visually cross gates which are not part of the gate may be displayed differently from the standard pipeline.

Unless you use `\pgfsetlayers` to add the layers `wires` and `behindwires` manually, `yquant` will automatically place the `wires` layer directly before `main`; then it will place `behindwires` directly before `wires`. This in particular ensures full compatibility with the `backgrounds` library: the background layer is still the very first in the layer list. In contrast to `pgfplots`, this automatism should work even if the `yquant` environment is placed within groups.

4.4 Overwriting the height and depth calculation

`yquant` automatically takes care of calculating the height and depth of all registers, so that their final vertical positions are chosen without overlap. This is almost always advisable, but it has some weaknesses:

- If you specify a multi-register gate, say, extending for three registers and this requires a certain height and depth, where should this be accounted for? `yquant` is able to handle these situations by first determining all heights and depths that can safely be attributed to individual registers. After that, it checks for all multi-register gates: Is the space from the top of the first to the bottom of the last register enough to hold the multi-qubit gate? If not, it evenly distributes the additional required space to all registers that are visually within the range of this multi-register. New in 0.4

  This will fail to produce good results (hopefully) only in two cases:

  - If you place `labels` on the gate, those are outside of the gate—and typically, either below or above. Hence, the additional extent stemming from them should *not* be equally distributed among all registers, but either to the height of the first or the depth of the last one. Currently, `yquant` is unable to detect this (and, considering the fact that you can place `labels` at any angle, this is not an easy problem to solve except for special cases).
  - If you make use of a discontiguous `init` gate with a large vertical extent, `yquant` will correctly allocate space as if the gate's content were placed in the vertical center. However, if there is no way to put the arch of the brace at the middle, as the register at this position is excluded from the gate, the content will be shifted—but only *after* calculating the extent. Hence, the automatically calculated vertical positions will be unsuitable.

- Sometimes, there is more space available than `yquant` thinks because you already discarded some wire. `yquant` does not keep track of whether the wires below or above a gate are actually visible at this position—which is not even be known at the time the gate command is issued, as horizontal positions are determined only in the drawing stage. Hence, you may choose to draw "within" the other, invisible wire.

In these certain special cases, you may want to turn off the automatic calculation for one particular gate. Note that you may then, depending on the situation, obtain results with overlapping gates. You can use the keys `/yquant/register/minimum height` and `/yquant/register/minimum depth` when declaring the relevant register to manually specify a larger desired value, but you have to experiment with regard to what this value is. New in 0.4

The global attribute `overlay` (conveniently overshadowing TikZ's `overlay` key, which should not be used for gates) can take the values

- `true` (default if no value given, combines `multi`, `height`, and `depth`),
- `multi` (short `m`),
- `height` (short `ht`, `h`), equivalent to `left` (short `l`) and `before` (short `bef`, `b`), which sound more meaningful for vertical and generic circuit orientations,
- `depth` (short `dp`, `d`), equivalent to `right` (short `r`) and `after` (short `aft`, `a`), which sound more meaningful for vertical and generic circuit orientations,
- `single` (short `s`, combines `height` and `depth`), and

- `false` (useless, default if attribute not given).

It disables the calculation of the selected vertical extent for this particular gate. (In fact, `multi`, `height`, `depth`, and `single` are subkeys that accept boolean values.)

5 Reference: Gates and operations

This section lists all operations yquant currently understands. It also details all arguments that can be given to customize the operation, apart from `name` and `overlay`, which are always available. Note that the `[value=<value>]` attribute can (and should) alternatively be given as a braced expression that follows the name of the register. Within `<value>`, unless specified differently, the macro `\idx` is always available and corresponds to the index of the current register in the list of targets. New in 0.4

5.1 addstyle

New in 0.1.2

Syntax: `setstyle <target>;`
This is an invisible pseudo-gate that immediately changes the TikZ style with which the register lines of all target registers are drawn. It adds to the styles that are already installed. Use `setstyle` to replace styles. It may not span multiple registers and does not allow for controls.

Possible attributes:

- `[value=<styles>]` (required)
  Denotes the new styles; this should be a string that could be passed to `\tikzset`.

5.2 align

Syntax: `align <target>;`
This is an invisible pseudo-gate that enforces all affected registers to share a common horizontal position for their next gate, which is determined by the largest position of all gates involved. It may not span multiple registers and does not allow for controls. The gate now always aligns the wires, i.e., if they are discarded directly after this gate, they will still discarded all at the same position. Changed in 0.4

Possible attributes: none

5.3 barrier

Syntax: `barrier <target>;`
This is a pseudo-gate that denotes some physical barrier that ensures execution with a specific timing; it is basically a visible version of the `align` gate, denoted by a vertical line. It may span multiple registers, but does not allow for controls. The style `/yquant/operators/every barrier` is installed.

Possible attributes: none

5.4 box

Syntax: `box <target> | <pcontrol> ~ <ncontrol>;`
This is a generic register of a rectangular shape that can be filled with arbitrary content. It may span multiple registers and allows for controls. The style `/yquant/operators/every box` is installed.
Possible attributes:

- `[value=<value>]`
  Denotes the content of the box.

5.5 cbit

Syntax: `cbit <name>[<len>];`
Declares a register of type `cbit`.
see qubit

5.6 correlate

New in 0.1.2

Syntax: `correlate <target>;`
This is a pseudo-gate that indicates a correlation (usually a Bell-state) present between the multi-registers involved. This gate should span multiple registers and does not allow for controls. The style `/yquant/operators/every wave` is installed.
Possible attributes: none

5.7 cnot

Syntax: `cnot <target> | <pcontrol> ~ <ncontrol>;`
This is a synonym for the `not` gate. Note that despite its name, controls are not mandatory and also here, the style `/yquant/operators/every not` is installed.

5.8 discard

Syntax: `discard <target>;`
This is an invisible pseudo-gate that changes the type of all target registers to `nobit`, i.e., no line will be drawn for them. This has effect already for the outgoing line of the last gate on the target registers. The gate may not span multiple registers and does not allow for controls. To change a register type on-the-fly into something different from `nobit`, use the `settype` pseudo-gate.

Changed in 0.1.2

Possible attributes: none

5.9 dmeter

Syntax: `dmeter <target>;`
This is a measurement gate, denoted by a "D" shape. It changes the type of all targets involved. It may span multiple registers, but does not allow for controls.
The style `/yquant/operators/every dmeter` is installed.
Possible attributes:

- `[value=<value>]`
  Allows to specify a text that will be included inside the gate, possible enlarging its width. For outside texts, use TikZ `labels` instead.

- `[type=<qubit|cbit|qubits>]`
  Allows to specify the type into which the affected targets are converted. Default is `cbit`.

5.10 h

Syntax: `h <target> | <pcontrol> ~ <ncontrol>;`
This is a Hadamard gate, $(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1|) \,/\, \sqrt{2}$, denoted by a rectangle that contains the letter H. It may not span multiple registers, but allows for controls.
The style `/yquant/operators/every h` is installed.
Possible attributes: none

5.11 hspace

Syntax: `hspace <target>;`
This is an invisible pseudo-gate that inserts a certain amount of white space into all target registers. It may not span multiple registers and does not allow for controls.
The gate now always has an effect, e.g., if the wire is discarded after this gate, it will still be extended by the given amount first. Changed in 0.4
Possible attributes:

- `[value=<dim>]` (required)
  Gives the amount of white space that is to be inserted. Must be a valid (nonnegative) TEX dimension.

5.12 init

Changed in 0.4

Syntax: `init <target>;`
This is a pseudo-gate that (re)initializes a registers to a given state. It may span

multiple registers, but does not allow for controls. The style /yquant/every label is installed. Note that this pseudo-gate, unlike all others, behaves differently if it is the first operation acting on a register: in this case, it does not increment the horizontal position, but uses the space available to the left; and the style /yquant/every initial label is installed additionally. Internally, creating a new register with some printed name is translated into the creation of an unnamed register, followed by application of this gate with the desired text.

Possible attributes:

- [type=<qubit|cbit|qubits>]
  Allows to specify the type into which the affected target registers are converted. Default is the type of the first target register that is different from nobit, or qubit if they all are nobit. The style /yquant/every <type> label is installed additionally.
- [value=<value>] (required)
  Denotes the label that is printed to the left of the wire.

5.13 inspect

New in 0.4

Syntax: inspect <target>;
This is a pseudo-gate that allows to print the current state of one or multiple registers within a circuit. It may span multiple registers, but does not allow for controls. The style /yquant/operators/every inspect is installed. Essentially, it is the same as an output gate that will be drawn immediately at the current position and not deferred until the end; hence, it also draws braces when used in a multi-register context. If this is not desired, use the text gate instead. New in 0.6

Possible attributes:

- [value=<value>] (required)
  Denotes the text that is to be printed.

5.14 iswap

New in 0.7.2

Syntax: iswap <targets> | <pcontrol> ~ <ncontrol>;
This is the two-qubit iSWAP gate $|00\rangle\langle 00| + \mathrm{i}\,|01\rangle\langle 10| + \mathrm{i}\,|10\rangle\langle 01| + |11\rangle\langle 11|$ that exchanges two qubits and conditionally adds phases. It is denoted by crosses within circles at the affected registers which are connected by a control line. It may span multiple registers (in fact, it should always span exactly two registers, though yquant does not enforce this), and it allows for controls. However, refrain from combining *multiple* two-qubit targets *together* with controls. The control line

- `[out]` or `[ancilla]` (required in subcircuits)
  see *qubit*

5.17 `not`

Syntax:

5.20 qubit

Syntax: `qubit <name>[<len>];`
Declares a register of type `qubit`. The `<name>` must be a self-chosen name for the register which was not previously used as a register name in this circuit (but names can be re-used in subcircuits). Names are case-insensitive. The register can be made into a vector register by specifying `<len>` (default `1`).

Possible attributes:

- `[after=<regname>]`
  If given, the register will start not at the left of the circuit but instead at the position at which the last gate in the register `<regname>` ended.

  This attribute may not be given in combination with `[in]` or `[inout]`.

- `[in]`, `[out]`, `[inout]`, or `[ancilla]` New in 0.2
  Default: `[ancilla]` for top-level circuits (do not change there); `[inout]` for subcircuits.

  Determines how a subcircuit interacts with its parent circuit.

  Registers declared with the `[ancilla]` attribute are available only to the subcircuit; they cannot be connected to an outside wire.

  Registers declared with the `[in]` or `[inout]` attribute will expect an outer wire of the same type to be present and will then be identical with this outer wire. Any changes applied to the wire within the subcircuit automatically also happen on the associated outer wire. If the attribute is `[in]`, the wire will automatically be discarded at the end of the subcircuit (and hence also in the outer circuit, where it may be re-initialized). This is different from applying the `discard` gate in that the wire will still extend until the end of the subcircuit and may thus receive proper `output`s.

  Registers declared with the `[out]` attribute will expect a discarded outer wire to be present, which will be initialized to a `qubit` at the beginning of the subcircuit, and from then on be identical with the outer wire.

- `[value=<value>]`
  Denotes the label that is printed to the left of the wire. If the value is omitted, the default is used (`/yquant/register/default name`, preinitialized to `\regidx`).

  Inside the value, `\reg` expands to `<name>`, `\len` expands to `<len>`, `\idx` expands to the current index within the vector register ($0 \leq$ `\idx` $<$ `<len>`), and `\regidx` expands to `\reg` if `<len>` is one, or to `\reg[\idx]` else.

5.21 qubits

Syntax: `qubits <name>[<len>];`
Declares a register of type `qubits`.
see qubit

5.22 setstyle

New in 0.1.2

Syntax: `setstyle <target>;`
This is an invisible pseudo-gate that immediately changes the TikZ style with which the register lines of all target registers are drawn. It replaces all previous styles. Use `addstyle` to accumulate styles. It may not span multiple registers and does not allow for controls.
Possible attributes:

- `[value=<styles>]` (required)
  Denotes the new styles; this should be a string that could be passed to `\tikzset`.

5.23 settype

New in 0.1.2

Syntax: `settype <target>;`
This is an invisible pseudo-gate that immediately changes the type of the targets registers, taking effect with the output line extending from the last drawn gate. It may not span multiple registers and does not allow for controls.
Possible attributes:

- `[value=<qubit|cbit|qubits>]` (required)
  Denotes the new type that is assigned to all registers. To change the type to `nobit`, use the `discard` pseudo-gate instead.

5.24 setwire

Removed in 0.4
Deprecated in 0.1.2

Use `settype` instead.
This gate is only available if a compatibility version before `0.4` is chosen.

5.25 slash

Changed in 0.4

Syntax: `slash <target>;`
This is a pseudo-gate used to denote that a single line actually represents multiple registers. It is drawn as a short slash through the line of the register. The style `/yquant/operators/every slash` is installed. Note that this gate will ignore

the /yquant/operator/minimum width key. With a compat key of 0.3 or earlier, the gate was special in that it did not advance the horizontal position on the wire, which allows to use it on only some of the wires without leading to a ragged start of subsequent gates. However, as yquant's default separation is not large enough to give a pleasant layout when the slash is squeezed in the initial separation, this was dropped as of version 0.4. Use the align gate after all slashes to get a better layout.

Possible attributes: none

5.26 subcircuit

New in 0.2

Syntax: subcircuit <target>;
This is a subcircuit gate which inserts independent quantum circuits at the current position within the circuit. It may span multiple registers, but is never split into contiguous slices. It allows for controls and may change the type of any target involved, depending on the particular subcircuit. The style /yquant/operators/every subcircuit is installed.

Possible attributes:

- frameless — New in 0.4
  This /yquant/operators/subcircuit/frameless style is activated with this shorthand.

- name mangling — New in 0.5
  This shorthand will pass the value of the attribute directly to the configuration key /yquant/operators/subcircuit/name mangling.

- seamless — New in 0.4
  The /yquant/operators/subcircuit/seamless style is activated with this shorthand (implies frameless).

- value=<subcircuit> (required)
  Denotes the content of the subcircuit. It is specified in the usual syntax of yquant. Note that, regardless of the outer environment, a subcircuit always implicitly uses the unstarred form, i.e., you must declare every register explicitly before its first usage. This is to make sure that the interface of the circuit, i.e., which registers are taken as input and/or output parameters and in which order, is not accidentally mistaken.

  The mapping between input and output registers is trivial for single-qubit uses. For multi-qubit uses, it works in the following way—in short, it matches

in visual order. You declare input and output registers by using the appropriate attributes on the `qubit`, `cbit`, `qubits` (or even `nobit`) gates. The list of all non-ancillas, from the topmost to the bottom-most, forms the list of parameter registers of the subcircuit. This is exactly the number of registers that must be supplied within one multi-qubit target. Also within the multi-qubit target, we sort all registers from the topmost to the bottom-most (in the order as they visually appear, not the order in which they are entered). Those two lists of equal length are then mapped $1:1$ to each other. Intermixing with ancillas is possible at every position and will lead to a vertical shift of the wires, until all registers, inner and outer, can be displayed flawlessly.

As subcircuits follow the same rules as ordinary circuits, it is possible to mix them with arbitrary TEX code, and also to access named gates within the subcircuit—but note that named gates in the outer circuit cannot be accessed (at least unless you play with the `name prefix` key in TikZ). In order to access inner nodes from the outer circuit, the subcircuit itself must be named; the inner nodes are then prefixed by the name of the subcircuit and a dash.

It is possible to nest subcircuits arbitrarily.

5.27 `swap`

Syntax: `swap <targets> | <pcontrol> ~ <ncontrol>;`
This is the two-qubit SWAP gate $|00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 01| + |11\rangle\langle 11|$ that exchanges two qubits. It is denoted by crosses at the affected registers which are connected by a control line. It may span multiple registers (in fact, it should always span exactly two registers, though `yquant` does not enforce this), and it allows for controls. However, refrain from combining *multiple* two-qubit targets *together* with controls. The control line will extend from the first to the last of all registers involved in the operation, so that it is impossible to discern visually which registers should actually be swapped. Using multiple swaps without controls in one operation is fine, as well as a single controlled swap. The style `/yquant/operators/every swap` is installed.

Possible attributes: none

5.28 `text`

New in 0.6

Syntax: `text <targets> | <pcontrol> ~ <ncontrol>;`
This is a pseudo-gate that allows to write some text within the circuit. It may span

multiple registers and allows for controls (though the situations in which controls make sense are pretty scarce). The style `/yquant/operators/every text` is installed. Contrary to the `inspect` gate, this gate will not draw curly braces in multi-register use. It basically corresponds to a `box` gate with suppressed drawing.

Possible attributes:

- `[value=<value>]` (required)
  Denotes the text that is to be printed.

5.29 x

Syntax: `x <target> | <pcontrol> ~ <ncontrol>;`
This is a Pauli σ_x gate $|0\rangle\langle 1| + |1\rangle\langle 0|$, denoted by a rectangle that contains the letter X. It may not span multiple registers, but allows for controls.
The style `/yquant/operators/every x` is installed.

Possible attributes: none

5.30 xx

Syntax: `xx <targets>;`
This is a symmetric flip gate, denoted by joined open squares. It should span multiple registers and it allows for controls. The same warnings as for the `swap` gate apply. The style `/yquant/operators/every xx` is installed.

Possible attributes: none

5.31 y

Syntax: `y <target> | <pcontrol> ~ <ncontrol>;`
This is a Pauli σ_y gate $-\mathrm{i}\,|0\rangle\langle 1| + \mathrm{i}\,|1\rangle\langle 0|$, denoted by a rectangle that contains the letter Y. It may not span multiple registers, but allows for controls.
The style `/yquant/operators/every y` is installed.

Possible attributes: none

5.32 z

Syntax: `z <target> | <pcontrol> ~ <ncontrol>;`
This is a Pauli σ_z gate $|0\rangle\langle 0| - |1\rangle\langle 1|$, denoted by a rectangle that contains the letter Z. It may not span multiple registers, but allows for controls.
The style `/yquant/operators/every z` is installed.

Possible attributes: none

5.33 zz

Syntax: `zz <targets>;`
This is a symmetric phase gate, denoted by joined filled circles. It should span multiple registers, but does not allow for controls. The same warnings as for the `swap` gate apply. The style `/yquant/operators/every zz` is installed.

Possible attributes: none

6 Examples

This section will contain lots of examples. On the left-hand side, the output is given, while the code to construct the example is on the right. All examples that are provided originate from the examples supplied with qasm, qcircuit, and quantikz. We will essentially follow their manuals example-by-example, which gives a nice comparison in how to achieve the given feature using these packages and yquant instead. All examples of course require inclusion of the yquant package with newest compatibility in the preamble, and some also require braket.

6.1 qasm documentation

The qasm documentation most often names the registers in the way $|\text{register}_{\text{index}}\rangle$. This can be achieved by writing

```
qubit {$\ket{<name>_{\idx}}$} <name>[<len>];
```

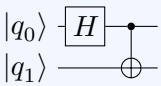
but if you want to realize this naming scheme for all circuits in your document, it is more convenient to say

```
\yquantset{register/default name=$\ket{\reg_{\idx}}$}
```

in the preamble, as is done here.

Note that yquant also directly supports the qasm syntax, see section 7.2. New in 0.3

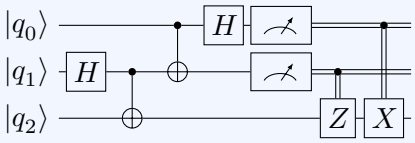
test1 (create an EPR pair)




```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[2];

      h q[0];
      cnot q[1] | q[0];
   \end{yquant}
\end{tikzpicture}
```

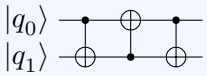
test2 (simple teleportation circuit)

Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[3];

      h q[1];
      cnot q[2] | q[1];
      cnot q[1] | q[0];
      h q[0];
      measure q[0-1];

      z q[2] | q[1];
      x q[2] | q[0];
   \end{yquant}
\end{tikzpicture}
```

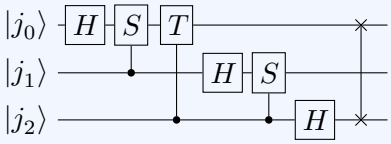
test3 (swap circuit)




```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[2];

      cnot q[1] | q[0];
      cnot q[0] | q[1];
      cnot q[1] | q[0];
   \end{yquant}
\end{tikzpicture}
```

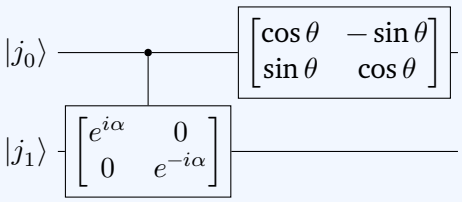
test4 (quantum fourier transform on three qubits)

Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant}
      qubit j[3];

      h j[0];
      box {$S$} j[0] | j[1];
      box {$T$} j[0] | j[2];
      h j[1];
      box {$S$} j[1] | j[2];
      h j[2];
      swap (j[0, 2]);
   \end{yquant}
\end{tikzpicture}
```

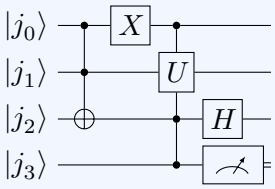
test5 (demonstrate arbitrary qubit matrix ops)




```
% \usepackage{amsmath}
\begin{tikzpicture}
   \begin{yquant}
      qubit j[2];

      box {$\begin{bmatrix}
         e^{i \alpha} & 0 \\
         0 & e^{-i \alpha}
      \end{bmatrix}$} j[1] | j[0];
      box {$\begin{bmatrix}
         \cos\theta & -\sin\theta \\
         \sin\theta & \cos\theta
      \end{bmatrix}$} j[0];
   \end{yquant}
\end{tikzpicture}
```

test6 (demonstrate multiple-qubit controlled single-q-gates)

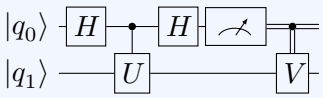
Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant}
      qubit j[4];

      cnot j[2] | j[0, 1];
      x j[0];
      box {$U$} j[1] | j[0, 2-3];
      h j[2];
      measure j[3];
   \end{yquant}
\end{tikzpicture}
```

test7 (measurement of operator with correction)

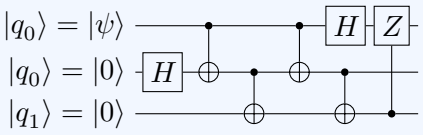
Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[2];

      h q[0];
      box {$U$} q[1] | q[0];
      h q[0];
      measure q[0];
      box {$V$} q[1] | q[0];
   \end{yquant}
\end{tikzpicture}
```

test8 (stage in simplification of quantum teleportation)

Updated in 0.4



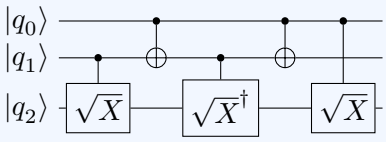

```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_0} = \ket\psi$}
       ↪  q[1];
      qubit {$\ket{q_{\idx}} = \ket0$}
       ↪  q[+2];

      h q[1];
      cnot q[1] | q[0];
      cnot q[2] | q[1];
      cnot q[1] | q[0];
      h q[0];
      cnot q[2] | q[1];
      z q[0] | q[2];
   \end{yquant}
\end{tikzpicture}
```

Note that we left out two Hadamards at the end.
Another way to provide various initial values in a single command is by performing case discrimination on `\idx`, for example in the following manner:

```
qubit {$\ket{q_{\idx}} = \Ifcase\idx\relax \ket\psi \Else \ket0 \Fi$}
 ↪  q[3];
```

In principle, all TEX conditionals that check against `\idx` need to be prefixed by `\protect`. If the `compat` key is at least `0.4`, `yquant` will make the commands `\Ifnum`, `\Ifcase`, `\Or`, `\Else`, `\Fi`, `\Unless` and `\The` available for use within gates; they correspond in a certain way to auto-`\protect`ed versions of the corresponding TEX primitives. Most likely, you will never need them inside values if not in the exact combination with `\idx`.

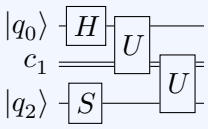
test9 (two-qubit gate circuit implementation of Toffoli)




```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[3];

      box {$\sqrt X$} q[2] | q[1];
      cnot q[1] | q[0];
      box {$\sqrt X^\dagger$} q[2] |
       ↪ q[1];
      cnot q[1] | q[0];
      box {$\sqrt X$} q[2] | q[0];
   \end{yquant}
\end{tikzpicture}
```

test10 (multi-qubit gates also demonstrates use of classical bits)

Alternative in 0.1.2
Updated in 0.1.1

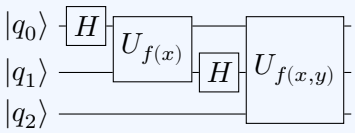



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_0}$} q;
      cbit {$c_1$} c;
      qubit {$\ket{q_2}$} q[+1];

      h q[0];
      box {$U$} (q[0], c);
      box {$S$} q[1];
      box {$U$} (c, q[1]);
   \end{yquant}
\end{tikzpicture}
```

Instead of a discontiguous vector register, we could also have used three scalar registers. The labels chosen for `qasm` do not fit well to the indices `yquant` assigns. We might also have used a three-register vector and used the `settype` pseudo-gate to immediately change the second register into a classical one, which would give indices matching the labels—but still, the registers would have a common name, which would make this a very unnatural approach.

test11 (user-defined multi-qubit ops)

Updated in 0.1.1

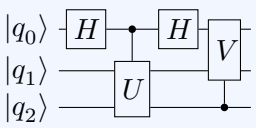



```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[3];

      h q[0];
      box {$U_{f(x)}$} (q[0, 1]);
      h q[1];
      box {$U_{f(x, y)}$} (q);
   \end{yquant}
\end{tikzpicture}
```

Here we used the fact that a vector register can also be addressed as a whole. Instead of `(q)`, we could have also written, e.g., `(q[0]-q[2])` or `(q[0-2])`, or enumerated all sub-registers in a comma-separated list.

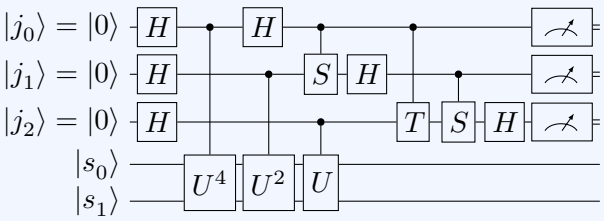
test12 (multi-qubit controlled multi-qubit operations)

Updated in 0.1.1



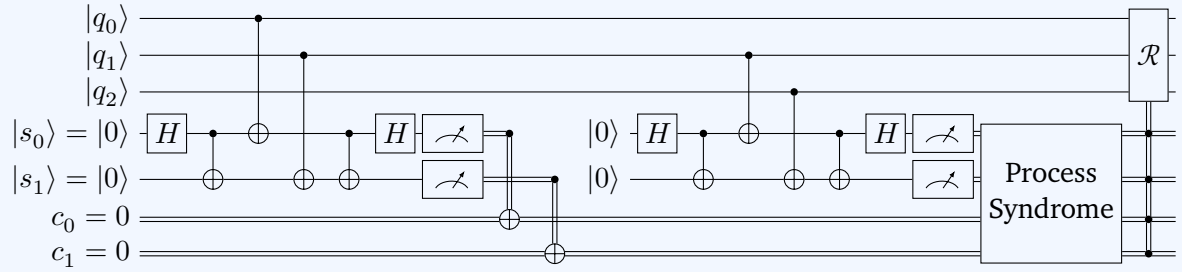

```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[3];

      h q[0];
      box {$U$} (q[1-2]) | q[0];
      h q[0];
      box {$V$} (q[0-1]) | q[2];
   \end{yquant}
\end{tikzpicture}
```

test13 (three-qubit phase estimation circuit with QFT and controlled-U)




```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{j_{\idx}} = \ket0$} j[3];
      qubit s[2];

      h j;
      box {$U^4$} (s) | j[0];
      box {$U^2$} (s) | j[1];
      box {$U$} (s) | j[2];
      h j[0];
      box {$S$} j[1] | j[0];
      h j[1];
      box {$T$} j[2] | j[0];
      box {$S$} j[2] | j[1];
      h j[2];
      measure j;
   \end{yquant}
\end{tikzpicture}
```

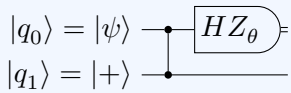
test14 (three-qubit FT QEC circuit with syndrome measurement)



```
\begin{tikzpicture}
   \begin{yquant}
      qubit q[3];
      qubit {$\ket{s_{\idx}} = \ket0$} s[2];
      cbit {$c_{\idx} = 0$} c[2];

      h s[0];
      cnot s[1] | s[0];
      cnot s[0] | q[0];
      cnot s[1] | q[1];
      cnot s[1] | s[0];
      h s[0];
      measure s;
      cnot c[0] | s[0];
      cnot c[1] | s[1];
      discard s; % to suppress wires extending until re-initialization

      init {$\ket0$} s;
      h s[0];
      cnot s[1] | s[0];
      cnot s[0] | q[1];
      cnot s[1] | q[2];
      cnot s[1] | s[0];
      h s[0];
      measure s;

      box {Process\\Syndrome} (s, c);
      box {$\symcal R$} (q) | s, c;
   \end{yquant}
\end{tikzpicture}
```

test15 (“D-type” measurement)

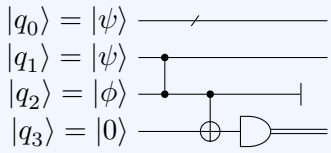
Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_0} = \ket\psi$} q;
      qubit {$\ket{q_1} = \ket+$} q[+1];

      zz (q);
      dmeter {$H Z_\theta$} q[0];
   \end{yquant}
\end{tikzpicture}
```

test16 (example from Nielsen paper on cluster states)

Updated in 0.4, 0.1.2, 0.1.1

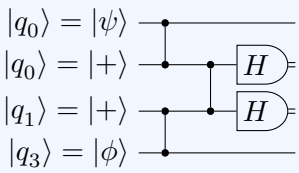



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_{\idx}} = \ket\psi$}
       ↪  q[2];
      qubit {$\ket{q_2} = \ket\phi$} q[+1];
      qubit {$\ket{q_3} = \ket0$} q[+1];

      zz (q[1], q[2]);
      align q;
      cnot q[3] | q[2];
      slash q[0];
      dmeter q[3];
      align q;
      [solid]
      barrier q[2];
      discard q[2];
   \end{yquant}
\end{tikzpicture}
```

We needed to include an `align` pseudo-gate to put the slash at the desired position. Usually, this would be sufficient to put the `cnot` and the `slash` gate directly under each other, as it is in the `qasm` example. However, the `slash` gate is special in that it does not need horizontal space and is put with only half of the usual operator separation into the circuit (for this reason, it can be put at the beginning of a wire without creating weird shifts with respect to the “unslashed” registers—it is put in the initial line that every wire even without an operation has). Hence, you should normally only use the `slash` gate as the very first gate in a circuit. To get the vertical stopper mark, we abuse a `barrier` on just a single wire and turn it from dashed to solid before `discard`ing.

test17 (example from Nielsen paper on cluster states)

Updated in 0.1.1

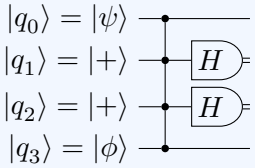



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_0} = \ket\psi$} q;
      qubit {$\ket{q_{\idx}} = \ket+$} q[+2];
      qubit {$\ket{q_3} = \ket\phi$} q[+1];

      zz q[(0-1), (2-3)];
      zz (q[1-2]);
      dmeter {$H$} q[1-2];
   \end{yquant}
\end{tikzpicture}
```

This example shows how the multi-qubit delimiter (the parenthesis) can even be used within indices.

test18 (multiple-control bullet op)

Updated in 0.1.2



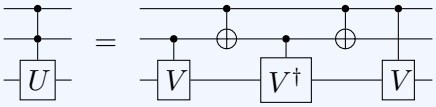

```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket{q_{\idx}} =
      ↪  \ket{\Ifcase\idx\relax \psi \Or + \Or +
      ↪  \Or \phi \Fi}$} q[4];

      zz (q);
      dmeter {$H$} q[1, 2];
   \end{yquant}
\end{tikzpicture}
```

This gate is probably a generalization of zz, $\mathbb{1} - 2\,|1\cdots1\rangle\langle1\cdots1|$, and indeed since version 0.1.2, we can use zz for this purpose. This time, we used the case distinction method in the initialization, as already alluded to before.

6.2 qcircuit documentation

For a better orientation, we use the same section headings as the qcircuit manual. The manual uses unnamed registers a lot; often, we will use the yquant* environment to make things more concise. As the qcircuit manual uses a bit larger separation between the operators than yquant's default, we globally say \yquantset{operator/separation=2mm}.

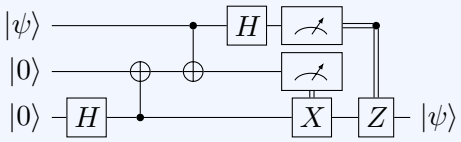
6.2.1 I. Introduction




```
% \useyquantlanguage{groups}
\begin{yquantgroup}
   \registers{
      qubit {} q[3];
   }
   \circuit{
      box {$U$} q[2] | q[0, 1];
   }
   \equals
   \circuit{
      box {$V$} q[2] | q[1];
      cnot q[1] | q[0];
      box {$V^\dagger$} q[2] | q[1];
      cnot q[1] | q[0];
      box {$V$} q[2] | q[0];
   }
\end{yquantgroup}
```

The best way to realize circuit equalities is with the help of groups language extension, which is documented in section 7.1.

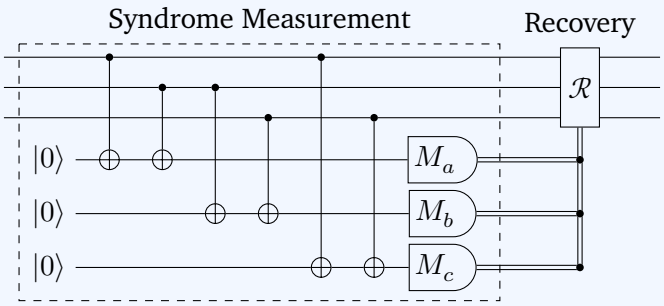
Updated in 0.5, 0.4, 0.1.1



Updated in 0.4

```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket\psi$} a;
      qubit {$\ket0$} b[2];

      h b[1];
      cnot b[0] | b[1];
      cnot b[0] | a;
      h a;
      align a, b;
      measure a;
      [direct control]
      measure b[0];

      x b[1] | b[0];
      z b[1] | a;

      discard a;
      discard b[0];
      output {$\ket\psi$} b[1];
   \end{yquant}
\end{tikzpicture}
```

Here, we see how to use a measurement as a direct output for the next controlled operation.

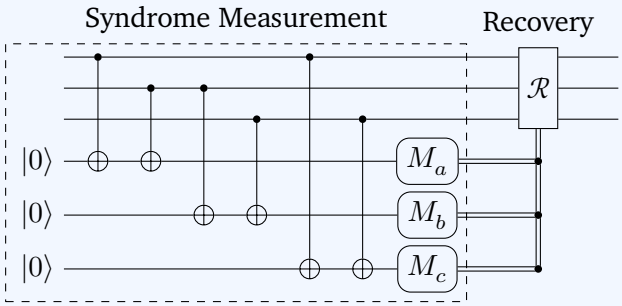


Updated in 0.4

```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant}
      qubit {} msg[3];
      nobit syndrome[3];

      [this subcircuit box style={dashed, "Syndrome Measurement"}]
      subcircuit {
         qubit {} msg[3];
         [out]
         qubit {$\ket0$} syndrome[3];

         cnot syndrome[0] | msg[0];
         cnot syndrome[0] | msg[1];
         cnot syndrome[1] | msg[1];
         cnot syndrome[1] | msg[2];
         cnot syndrome[2] | msg[0];
         cnot syndrome[2] | msg[2];

         dmeter {$M_{\symbol{\numexpr`a+\idx}}$} syndrome;
      } (msg[-2], syndrome[-2]);

      ["Recovery"]
      box {$\mathcal R$} (msg) | syndrome;
      discard syndrome;
   \end{yquant}
\end{tikzpicture}
```

The example demonstrates how to put a description next to a gate. In general, those descriptions should be realized using the TikZ feature `label`. Using the TikZ library `quotes`, the label is most easily specified. Since the label is not part of the valid arguments and also cannot be found in the `/yquant` path, it is automatically passed to `/yquant/operator style`.
To enclose a part of the circuit by a rectangle, we use a subcircuit. We define the incoming `qubit`s in the outer circuit, they will have the default attribute `[inout]`; the syndrome registers, which are created only in the subcircuit enter as `nobit`s and consequently have the `[out]` attribute. It is important to note that both the `dashed` style as well as the `label` (here with quoted syntax) are specified only inside `/yquant/this subcircuit box style`. This ensures that they are not also attached to every single gate in the subcircuit.
Then we see how to apply an operation to multiple registers in parallel while using the `\idx` macro to still give them a different text. Since `\idx` gives a numerical index (zero-based), we exploit the ASCII code (actually, this document

is compiled in Unicode mode…) to turn this into a letter.
Note that it could have become necessary to pass the `overlay` attribute to the recovery gate, as it is a multi-register gate with a `label`, meaning that `yquant` cannot reliably distribute its total vertical extent over its constituent registers. However, as the $\mathcal{R}$ together with the label in total were not higher than the three-qubit gate would have been anyway, this was not necessary here. In generally, don't use `overlay` unless necessary; maybe a future version will even be able to handle the more difficult cases better.
Finally, we will give a similar circuit by using the TikZ interface instead of subcircuits, this time also showing how we can change the shape of the measurement gate to one as in the `qcircuit` manual:



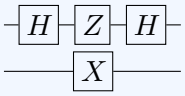
```
% \usetikzlibrary{fit, quotes}
\begin{tikzpicture}
   \begin{yquant}
      qubit {} msg[3];
      [name=inits]
      qubit {$\ket0$} syndrome[3];

      [name=scnot0]
      cnot syndrome[0] | msg[0];
      cnot syndrome[0] | msg[1];
      cnot syndrome[1] | msg[1];
      cnot syndrome[1] | msg[2];
      cnot syndrome[2] | msg[0];
      cnot syndrome[2] | msg[2];
      [name=smeas, shape=yquant-rectangle, rounded corners=.45em]
      dmeter {$M_{\symbol{\numexpr`a+\idx}}$} syndrome;
      ["Recovery"]
      box {$\symcal R$} (msg) | syndrome;
      discard syndrome;
   \end{yquant}
   \node[draw, dashed, fit=(inits-2) (scnot0-p0) (smeas-2), "Syndrome
    ↪  Measurement"] {};
\end{tikzpicture}
```

We name several elements that visually form the enclosing rectangle; then, we use the TikZ library `fit` to put a node around them all. Any gate can be given a custom shape; here, we use a `yquant-rectangle`, which is the analogue to TikZ's `rectangle` and thus supports the standard `rounded corners` style.

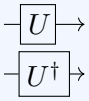
6.2.2 IV. Simple Quantum Circuits


```
\begin{tikzpicture}
   \begin{yquant*}
      x q;
   \end{yquant*}
\end{tikzpicture}
```

A. Wires and gates




```
\begin{tikzpicture}
   \begin{yquant*}
      h a;
      align a, b;
      z a;
      x b;
      h a;
   \end{yquant*}
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant*}
      box {$U$} a;
      box {$U^\dagger$} b;
      setstyle {->} -;
   \end{yquant*}
\end{tikzpicture}
```

yquant allows to change wire styles by means of the setstyle and addstyle pseudo-gates. Here, we use the gate on all wires in order to set an arrow style. Note that arrowheads are actually very special in two respects:

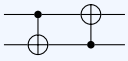
- yquant draws continuous wires for as long as possible. In this example, the wire path extends from the very left to the end of the circuit; yquant does not draw a wire *to* the gate and then a separate one *from* the gate to the next or the end. The only way to force yquant to draw multiple wires is to change the wire style or type mid-circuit. For example, by saying `addstyle {} -;`, all wire paths will be separated at the current position, which *in theory* allows to draw arrowsheads on intermediate wires.

Changed in 0.1.2
Updated in 0.1.2

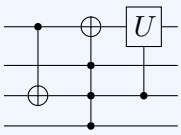
- *In practice*, this will not work due to the clipping commands that yquant installs. Every wire extends from the center of the left to the center of the right gate, and the gate's shape acts as a clipping path. Consequently, though the arrowhead is drawn, it is actually drawn at the center of the gate instead of the `west` anchor and then clipped away (unless the gate is small, in which case you might still see some fragments of the arrowhead).

Thus, it is currently not possible to use arrowheads on intermediate wires. If you really need to do this (say, for only a single gate), you may experiment with the TikZ `shorten` keys, which allow you to manually reduce the length of the wire, but the amount of reduction must be hand-computed for every gate. If you need this more often, consider filing a feature request.

B. CNOT and other controlled single qubits gates



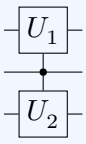
```
\begin{tikzpicture}
   \begin{yquant*}
      cnot a[1] | a[0];
      cnot a[0] | a[1];
   \end{yquant*}
\end{tikzpicture}
```



```
\begin{tikzpicture}
   \begin{yquant*}
      cnot q[2] | q[0];
      cnot q[0] | q[1-3];
      box {$U$} q[0] | q[2];
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.1.1

C. Vertical wires



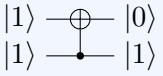
```
\begin{tikzpicture}
   \begin{yquant*}
      box {$U_{\The\numexpr\idx+1}$} q[0, 2] | q[1];
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.4, 0.1.1

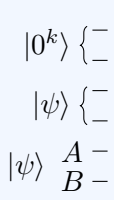
There is no direct support for this construction, but as with the initialization of a vector registers, `yquant` allows to access the macro `\idx` within an operator `value`. This macro follows the same rules as the name suffix, i.e., it assigns indices (zero-based) to the target registers in top-to-bottom order, regardless of which order was specified in the target list. Since we instead want a one-based subscript, we need to add one. Note that if you want to output `\idx` directly or within an unexpandable expression, you don't need to take any

action. However, here, \the is expandable; and since yquant needs to process all its output twice (first in order to determine the vertical spacing, second to actually typeset), you must manually take care that the command is *not* expanded prematurely by saying \protect\the instead, for which yquant, with a compat setting of at least 0.4, provides the shorthand \The. Had we used the plain TEX \the instead, the subscript would have been "1" for both operators.

D. Labeling input and output states



```
\begin{tikzpicture}
   \begin{yquant*}
      qubit {$\ket1$} q[2];
      cnot q[0] | q[1];
      output {$\ket\idx$} q;
   \end{yquant*}
\end{tikzpicture}
```



```
% \usetikzlibrary{calc}
\begin{tikzpicture}
   \begin{yquant*}
      {
         \yquantset{every multi label/.style={every
          ↪ node/.style={anchor=east, midway}}}
         init {$\ket{0^k}$} (a[-1]);
      }
      init {$\ket\psi$} (b[-1]);
      [name=cinit]
      qubit {\Ifcase\idx\relax$A$\Or$B$\Fi} c[2];
      \node[anchor=east] at
       ↪ ($(cinit-0.west)!.5!(cinit-1.west)$) {$\ket\psi$};
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.4, 0.1.1

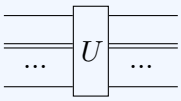
Here, three different styles for the initialization of multi-qubit labels are used. The second one (using a curly brace) corresponds to the default. It is overwritten for the first qubit, and to make this modification local, this is done in a group. The third qubit pair uses an overall label and additionally individual labels on the lines. The recommended way to do this starting from version 0.4 is to add the "special" label by means of a TikZ command.

6.2.3 V. More Complicated Circuits: Multiple Qubit gates and Beyond

A. Multiple qubit gates


```
\begin{tikzpicture}
   \begin{yquant*}
      box {$U^\dagger$} (a[-2]);
   \end{yquant*}
\end{tikzpicture}
```

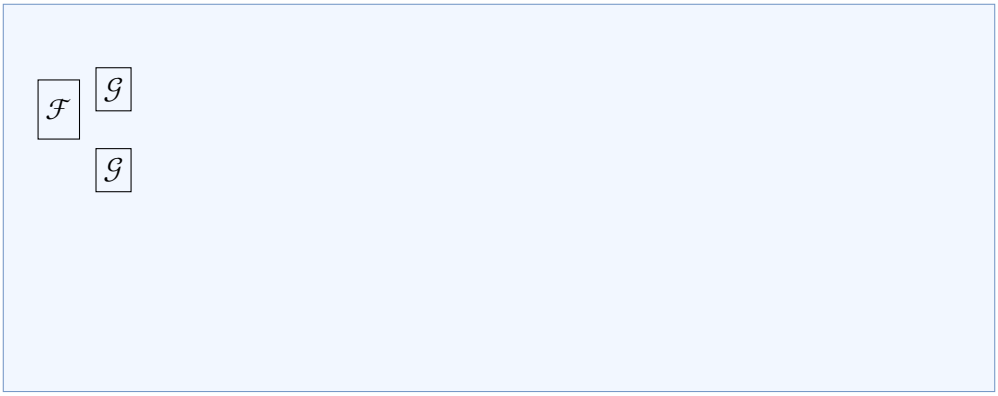
Updated in 0.1.1

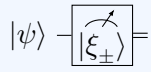



```
\yquantdefinebox{dots}[inner sep=0pt]{$\dots$}
\begin{tikzpicture}
   \begin{yquant}
      qubit {} a;
      cbit {} b;
      [register/minimum height=0pt, register/minimum
      ↪ depth=0pt]
      nobit ellipsis;
      qubit {} c;

      dots ellipsis;
      box {$U$} (a, b, ellipsis, c);
      dots ellipsis;
   \end{yquant}
\end{tikzpicture}
```

Updated in 0.4, 0.1.1

This demonstrates how a register of type `nobit` might even be useful if the register is never used and no subcircuits are involved. Note how we overwrite the default minimum height and depth setting for this register only. Additionally, we for the first time define our own gate, which we call `dots`. As we define our own style, it does not inherit from `/yquant/operators/every box`; hence, we only need to overwrite the `inner sep` coming from TikZ's defaults.

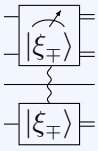

$\mathcal{F}$
$\mathcal{G}$
$\mathcal{G}$



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket\psi$} q;

      measure {$\ket{\xi_\pm}$} q;
   \end{yquant}
\end{tikzpicture}
```

```
\begin{tikzpicture}
   \begin{yquant*}
      dmeter {Bell} (a[0, 1]);
      discard a;
   \end{yquant*}
\end{tikzpicture}
```

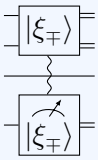
Updated in 0.1.1



```
\begin{tikzpicture}
   \begin{yquant*}
      measure {$\ket{\xi_\mp}$} (a[-1, 3]);
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.1.2, 0.1.1

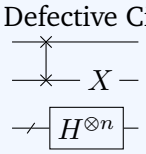
Multi-qubit gates (including measurements) on non-adjacent registers are properly supported. As explained in section 2.4, there is one main and multiple subordinate gate in such a discontiguous multi-qubit operation (though at the moment, the `measure` gates with text is the only gate that makes this distinction). In our case, the main part contains the measurement symbol and the text, while the subordinate gates only contain the text. By default, `yquant` uses the first slice as main part, but you may influence this by preceding what you want to be “main” by a star:



```
\begin{tikzpicture}
   \begin{yquant*}
      measure {$\ket{\xi_\mp}$} (a[-1, *3]);
   \end{yquant*}
\end{tikzpicture}
```

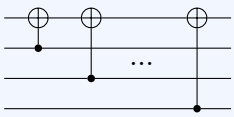
C. Non-gate inserts, forcing space, and swap

Updated in 0.6, 0.1.1




```
\begin{tikzpicture}
   \begin{yquant*}
      [name=sw]
      swap (a[0-1]);
      text {$X$} a[1];
      slash b;
      box {$H^{\otimes n}$} b;
      \node[anchor=199] at (sw-0.north) {Defective
      ↪  Circuit};
   \end{yquant*}
\end{tikzpicture}
```

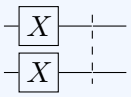
Here, the intermediate text was inserted by using a `text` gate; before version 0.6, this would have to be done by a `box` gate with `[draw=none]` attribute. Another way would be to use an `init` command, although this is semantically wrong (probably).




```
\begin{tikzpicture}
   \begin{yquant*}
      qubit {} a;
      [name=ypos]
      qubit {} b[3];

      cnot a | b[0];
      [name=left]
      cnot a | b[1];
      hspace {7mm} -;
      [name=right]
      cnot a | b[2];
   \end{yquant*}
   \path (left |- ypos-0) -- (right |- ypos-1)
   ↪  node[midway] {$\dots$};
\end{tikzpicture}
```

Note how the register range `-` was used to denote all registers. We positioned the dots by first naming the relevant registers, so that the vertical position is at the coordinates `ypos-0` and `ypos-1`; and then, we also named the `cnot` gates, so that we are able to discern the horizontal position.

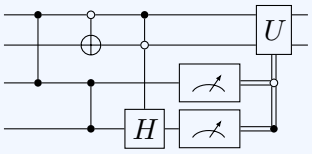
D. Barriers




```
\begin{tikzpicture}
   \begin{yquant*}
      x a[0, 1];
      barrier (a);
   \end{yquant*}
\end{tikzpicture}
```

Now the qcircuit manual lists three circuits with barriers at different positions. They cannot be drawn with yquant; however, since neither of them is a valid circuit (no indication whether the control is positive or negative), this is of no concern.

E. How to control anything




```
\begin{tikzpicture}
   \begin{yquant*}
      zz (a[0, 2]);
      cnot a[1] ~ a[0];    {
      zz (a[2, 3]);
      h a[3]
```

6.2.4 VI. Bells and Whistles: Tweaking Your Diagram to Perfection

For options how to configure the circuits, refer to section 3.

A. Spacing

Updated in 0.1.1


```
\begin{tikzpicture}
   \begin{yquant*}
      swap (a[0, 1]);
      box {$T^\dagger$} a[2] | a[1];
      swap (a[0, 1]);
   \end{yquant*}
\end{tikzpicture}
```

B. Labeling

Updated in 0.6, 0.4

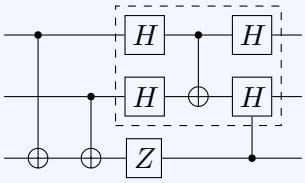

```
\begin{tikzpicture}
   \begin{yquant}[every initial
   ↪  label/.style={anchor=south east, yshift=1mm},
   ↪  every post measurement control=direct]
      qubit {\rlap{\hskip2mm $a$}} a;
      qubit {\rlap{\hskip2mm $b$}} b;
      nobit out;
      hspace {5mm} -;

      measure b;
      text {$B$} out | b;
      measure a;
      text {$A$} out | a;
      discard -;
   \end{yquant}
\end{tikzpicture}
```

We support measurements with vertical outputs, but only if they replace the positive control of some action. Here, we fake this behavior by introducing an invisible register at the bottom, which will contain the outputs. Note that if we were to give style options to the `text`s, they would also affect the measurements. The reason for this is that internally, the measurement will be nested within the same scope that draws the `text`—so the options given to the `text` will be inherited by the `measure`ment. As an operator style overwrites default styles, this will also apply to the measurements. Hence, to circumvent this, we would need to *revert* the options as attributes to the `measure`ments, even if the

reverted option was already included in their native style.
Repositioning the initial labels needs some care and manual fine-tuning.

C. Grouping

Updated in 0.4, 0.1.1



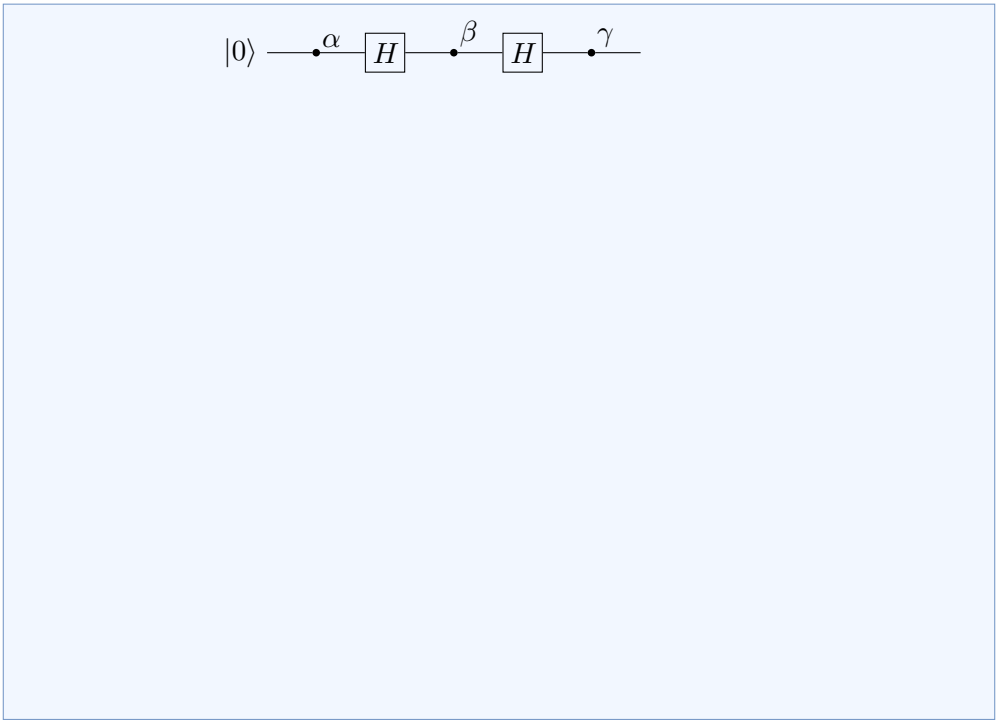

```
% \usetikzlibrary{fit}
\begin{tikzpicture}
   \begin{yquant*}[register/separation=3mm]
      cnot a[2] | a[0];
      cnot a[2] | a[1];
      [name=left]
      h a[0, 1];
      z a[2];
      cnot a[1] | a[0];
      [name=righttop]
      h a[0];
      [name=rightbot]
      h a[1] | a[2];

      hspace {2mm} -;
   \end{yquant*}
   \node[draw, dashed, fit=(left-0) (left-1) (righttop) (rightbot-0)] {};
\end{tikzpicture}
```

Note that `\begin{yquant*}` must not be followed by a line break (unless masked by `%`) if options follow. Also note that here, we cannot make use of a subcircuit due to the very last control, which would then control an inner gate of said subcircuit—but they are not exposed.

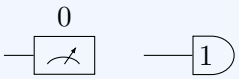
6.3 quantikz documentation

Again, our section headings will be the same as in the quantikz manual. And since quantikz also has even more space between the gates, we globally say `\yquantset{operator/separation=4mm}`.

6.3.1 II. A single wire



A. Measurements

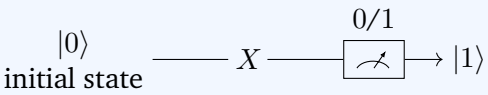



```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}
      ["$0$"]
      measure a;
      discard a;

      init {} a;
      dmeter {$1$} a;
      discard a;
   \end{yquant*}
\end{tikzpicture}
```

Other measurement shapes are not supported at the moment.

B. Wires and arrows

Updated in 0.6, 0.1.2

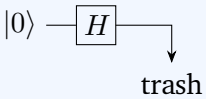



```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant}[operator/separation=1cm, every label/.append
   ↪  style={align=center}]
      qubit {$\ket0$\\initial state} a;

      text {$X$} a;

      ["$0$/$1$", type=qubit]
      measure a;

      addstyle {->} a;
      output {$\ket1$} a;
      \yquantset{operator/separation=5mm}
   \end{yquant}
\end{tikzpicture}
```

This example demonstrates how to instruct the `measure` gate to use a different output type than the standard `cbit`.
In general, any macros that are used within a TikZ path or a `yquant` operation must not be fragile, or must be preceded with `\protect`. In this example, `\\` is a robust command (at least in newer kernels), so protection is not required.

Since it may occur quite frequently that yquant is used within a center environment or in \centering mode (in which \\ is still fragile), yquant takes care of this (it actually robustifies \@centercr, which is the meaning of \\ in these surroundings—and which is now incorporated into the LaTeX kernel as of June 2021).
In order to change the style of an individual wire, we use addstyle. To make the final line shorter, we change the operator separation by issuing \yquantset at the end.

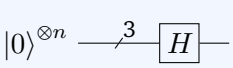
New in 0.4




```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket0$} q;
      [name=h]
      h q;
      discard q;
      \path[/yquant/every wire, /yquant/every qubit
       ↪ wire, -Stealth] (h) -| ++(1cm, -.5cm)
       ↪ node[below] {trash};
   \end{yquant}
\end{tikzpicture}
```

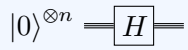
Here, we use an ordinary \path command to reproduce the “trash” line. This time, we chose to use the appropriate styles as yquant itself would do it instead of just saying \draw without the options, which would also have worked.

Updated in 0.4



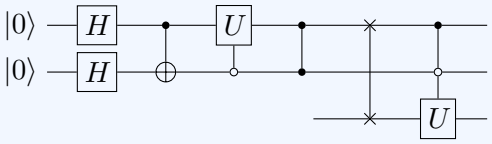

```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}
      qubit {$\ket0^{\otimes n}$} a;
      ["north east:3" {font=\protect\footnotesize,
       ↪ inner sep=0pt}]
      slash a;
      h a;
   \end{yquant*}
\end{tikzpicture}
```

Again, you see an example of how some commands need to be \protected when used in yquant options, and that you can indeed exploit all features of the quotes library.



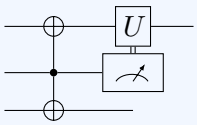

```
\begin{tikzpicture}
   \begin{yquant}
      qubits {$\ket0^{\otimes n}$} a;
      h a;
   \end{yquant}
\end{tikzpicture}
```

6.3.2 III. Multiple Qubits




```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket0$} a;
      qubit {$\ket0$} b;

      h a, b;
      cnot b | a;
      box {$U$} a ~ b;
      zz (a, b);

      [after=a]
      qubit {} c;

      swap (a, c);
      box {$U$} c | a ~ b;
   \end{yquant}
\end{tikzpicture}
```

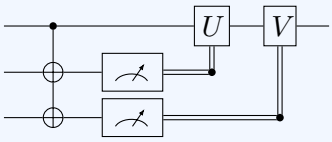
This example demonstrates the use of the `after` argument that instructs the register creation to begin the register only after the current position of another register that already exists. Note that this argument will always make the wire begin *at the right end* of the last gate of the referenced register; however, if—as is the case here—this gate is shorter than `/yquant/operator/minimum width`, this might not coincide with the visual right end.




```
\begin{tikzpicture}
   \begin{yquant*}
      [name=c]
      cnot a[0, 2] | a[1];
      [name=m, direct control]
      measure a[1];
      discard a[2];
      box {$U$} a[0] | a[1];
      \path[/yquant/every wire, /yquant/every qubit wire] (c-1) --
      ↪  (m.center |- c-1);
      discard a[1];
   \end{yquant*}
\end{tikzpicture}
```

Here, we manually extended the wire on the last register. We could instead have performed an `align` gate before the discarding process, then, the wire line would have been extended by yquant; but since `align` aligns at the *right end* as opposed to the center of the gate, the wire line would have been a bit longer. Still, this TikZ wire is inferior to a wire drawn by yquant, as it does not use clippings: the connection with the `cnot` gate may not be accurate; in particular, if the wire is of a different color or if you need to draw classical or bundle wires, the connection will become unpleasant.

Updated in 0.4, 0.1.2, 0.1.1



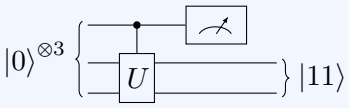

```
\begin{tikzpicture}
   \begin{yquant*}
      cnot a[1, 2] | a[0];
      measure a[1], a[2];
      box {$U$} a[0] | a[1];
      box {$V$} a[0] | a[2];
      discard a[1]-;
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.1.1

yquant doesn't offer anything comparable to the new `\ctrlbundle` command; and as the bundle lines are spaced much more tightly in yquant, this would not really make sense.

6.3.3 IV. Operating on many Qubits

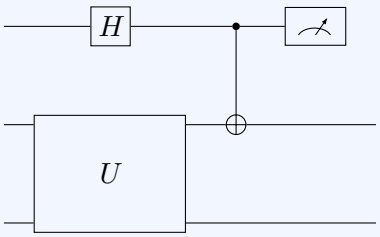
Updated in 0.1.1




```
\begin{tikzpicture}
   \begin{yquant*}
      init {$\ket0^{\otimes3}$} (a[-2]);

      box {$U$} (a[1-2]) | a[0];
      measure a[0];
      discard a[0];
      output {$\ket{11}$} (a[1-2]);
   \end{yquant*}
\end{tikzpicture}
```

Multi-qubits inputs are possible using the `init` command. The text assigned to a register declaration is always for an individual register.

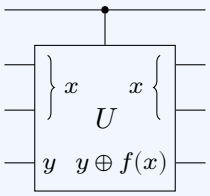
Updated in 0.4




```
\begin{tikzpicture}
   \begin{yquant*}[register/minimum
   ↪  height=6mm, register/minimum
   ↪  depth=6mm]
      hspace {7.5mm} a;
      h a;
      hspace {7.5mm} a;
      [x radius=1cm]
      box {$U$} (b, c);
      cnot b | a;
      measure a;
      discard a;
   \end{yquant*}
\end{tikzpicture}
```

`yquant` does not use a grid layout: operators are stacked next to each other. Therefore, there is no automatic centering of a column, though it could be emulated using hand-crafted `hspace` commands, as was done here (the Hadamard gate uses the `/yquant/operator/minimum width`, which is `5mm`, while the large box has a width of `2cm`, so that we need two `7.5mm` spacings at the end, as the `hspace` pseudo-gate only inserts exactly the space you give, but not additional [twice] `/yquant/operator/separation`, as would be the case for a hypothetical zero-width gate). In fact, we don't even need the second `hspace`, since the two-qubit `cnot` will automatically enforce correct alignment.

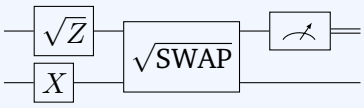
Updated in 0.4




```
\begin{tikzpicture}
   \begin{yquant}[register/separation=3mm, every nobit output/.style={}]
      qubit {} a[4];
      [every inspect/.append style={outer xsep=0pt}, operator/minimum
       ↪  width=0pt, font=\footnotesize, name=sub]
      subcircuit {
         \yquantset{operator/separation=0pt}
         qubit {} x[2];
         qubit {} y;
         discard -;

         inspect {$x$} (x);
         [inner xsep=0pt]
         inspect {$y\vphantom f$} y;

         [shape=yquant-init, decoration={mirror}]
         inspect {\hskip4mm $x$} (x);
         [shape=yquant-init, inner xsep=0pt]
         inspect {$y \oplus f(x)$} y;
      } (a[1-3]) | a[0];
      \node at (sub) {$U$};
      settype {qubit} a;
   \end{yquant}
\end{tikzpicture}
```

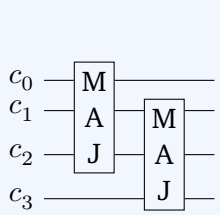
We use quite some tricks to achieve such a layout. We use a subcircuit as a container and `inspect` gates to indicate the inputs and output states accurately. As those are ordinary gates, we reset `/yquant/operator/minimum width`, so that the "y" indication is not too long. Additionally, they are usually meant to be used within a circuit, i.e., they have an additional margin denoted by the `outer xsep`, which we also remove. Then, within the subcircuit, we reset the `/yquant/operator/separation`, which would insert additional whitespace at the beginning. It is important to do this within the subcircuit and not as an attribute; else, we would also remove the outer lines going into the subcircuit. Initial or final `inspect` gates without a brace do not really need the separation between brace tip and text (`inner xsep`), so we also remove it. The output gates should have their braces and separations at the other side, which cor-

responds to changing their shape from `yquant-output` to `yquant-init` and mirroring the decoration (as in `/yquant/every label`). To get the desired right-alignment, we hand-tailor an `\hskip` that enlarges the upper output label—automatic alignments would not work here: putting the two `inspect`s together with a case distinction on `\idx` would center them; using `output`s would left-align them.
Finally, we have to deal with the caption of the gate, which should be absolutely centered with respect to the subcircuit and not have any influence on the spacing—so we just insert it retrospectively as an ordinary TikZ node.
Also note the use of `discard` and `settype` since we needed wires before and after the subcircuit, which must match the inner wires in type, but we actually do not want to have inner wires.




```
\begin{tikzpicture}
   \begin{yquant*}
      box {$\sqrt Z$} a;
      box {$X$} b;
      box {$\sqrt{\mathrm{SWAP}}$} (a, b);
      measure a;
   \end{yquant*}
\end{tikzpicture}
```

This time, we did not artificially discard the lines.

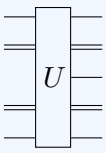



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$c_{\idx}$} c[4];
      box {M\\A\\J} (c[-2]);
      box {M\\A\\J} (c[1-]);
   \end{yquant}
\end{tikzpicture}
```

Updated in 0.4, 0.1.1

Notice here that the vertical spacing is uneven. `yquant` realizes that the minimal vertical spacing will not be enough to account for the multi-qubit boxes. However, when it tries to adjust positions accordingly so that the last gate fits, this will of course not change anything for the first wire, which is not contained in the gate. After having increased the spacing, `yquant` realizes that this already was enough to accomodate for the first gate, so no further action is taken. In order to get a more even spacing, just increase `/yquant/register/minimum height` and `/yquant/register/minimum depth`.

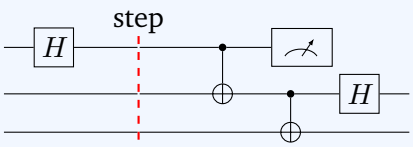
A. Different connections




```
\begin{tikzpicture}
   \begin{yquant}[register/default name=]
      qubit a;
      cbit b;
      nobit c;
      cbit d;
      qubit e;
      box {$U$} (-);
      settype {qubit} c;
   \end{yquant}
\end{tikzpicture}
```

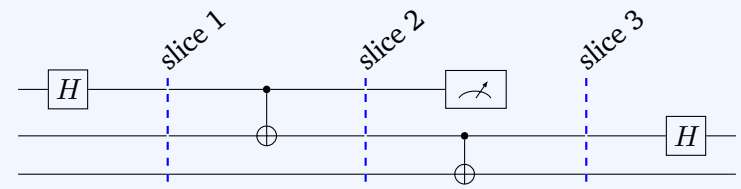
This example demonstrates the declaration of a non-existing register and the `settype` pseudo-gate that acts as a zero-width, no-content `init` gate.

6.3.4 V. Slicing

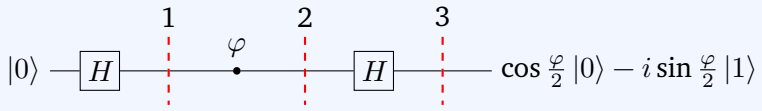



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {} a[3];
      h a[0];
      [red, thick, label=step]
      barrier (a);
      cnot a[1] | a[0];
      measure a[0];
      discard a[0];
      cnot a[2] | a[1];
      h a[1];
   \end{yquant}
\end{tikzpicture}
```

There is nothing like a `slice all` keyword, as yquant's underlying layout is not grid-based. Changing the style of slice captions simply means providing label options. This time, we used the `label` key instead of the shorter syntax provided by the `quotes` library, which is of course also possible.




```
% \usetikzlibrary{quotes}
\begin{tikzpicture}[every label/.style={rotate=40, anchor=south west}]
   \begin{yquant}[operators/every barrier/.append style={blue, thick}]
      qubit {} a[3];
      h a[0];
      ["slice 1"]
      barrier (-);
      cnot a[1] | a[0];
      ["slice 2"]
      barrier (-);
      measure a[0];
      discard a[0];
      cnot a[2] | a[1];
      ["slice 3"]
      barrier (-);
      h a[1];
   \end{yquant}
\end{tikzpicture}
```



Updated in 0.4

```
% \usetikzlibrary{quotes}
\begin{tikzpicture}[label distance=4mm]
   \begin{yquant}[operators/every barrier/.append style={red, thick,
   ↪  shorten <= -4mm, shorten >= -4mm}]
      qubit {$\ket0$} a;
      h a;
      ["1"]
      barrier a;
      phase {[label distance=0pt]$\varphi$} a;
      ["2"]
      barrier a;
      h a;
      ["3"]
      barrier a;
      output {$\cos\frac\varphi2 \ket0 - i\sin\frac\varphi2 \ket1$} a;
   \end{yquant}
\end{tikzpicture}
```

Usually, the `shorten` keys do not have any effect on `yquant` operations, since the latter are all made up of nodes. However, the `yquant-line` shape explicitly takes care of correctly handling them. It is the only one that does so. Since barriers usually end quite closely to the wires—and the default `dashed` style may make this worse—the `shorten`ing may often prove useful. Note that if the barriers are enlarged by means of negative `shorten`ings, this will not affect the bounding box and you must take care of appropriately shifting labels. The internal register height calculations might be inconsistent for multi-register barriers with `shorten`ing: While `yquant` takes care of enlarging the registers so that there is enough space for placing the `barrier` with its *original* (single-register) height, it actual height of the registers is only known at the second stage in calculation; but applying `shorten`ings after this stage would require another iteration of height calculation. Hence, multi-register barriers that are enlarged by a lot will probably look bad unless you add manual spacing to the appropriate registers. Also note that we used much larger magnitudes in order to achieve a similar appearance as in `quantikz`. To avoid that the large distance also affects the `phase` gate badly, we locally reset the distance; for this, there are two ways. The easiest one is to make use of the fact that the value of the `phase` gate is passed directly as `label` argument, so that we can locally reset the distance. The other possibility would be to write

```
{
   \yquantset{/tikz/label distance=0pt}
   phase {$\varphi$} a;
}
```

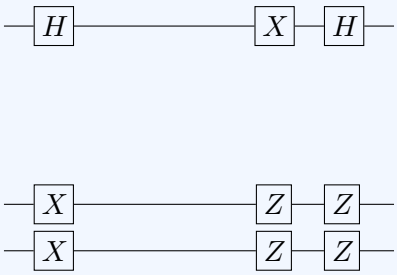
since due to the aforementioned lack of support for a style that sets the options in TikZ, we must manually use a (grouped) `\yquantset` instruction for this. Note that whenever you change a TikZ style in a `yquant` environment, use the `\yquantset` macro, *not* `\tikzset` or `\pgfkeys`. Not only will the latter two not automatically restart the parser (so that you would have to issue `\yquant` after their use), but `yquant` has to process all its content twice in order to properly determine the register height. Only `\yquantset` will be properly captured and re-issued at the correct position when the content is actually typeset. Had we written `\tikzset{label distance=0pt} \yquant`, no effect at all would have been visible, since this command would only have taken effect in the first (invisible) round when `yquant` determines heights.

`yquant` does not provide a mechanism for vertical labels, but you may of course just insert line breaks at appropriate positions (and set the `align` property of the labels).

6.3.5 VI. Spacing

A. Local adjustment

Updated in 0.4



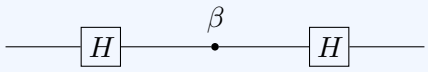

```
\begin{tikzpicture}
   \begin{yquant}[register/default name=]
      [register/minimum depth=2cm]
      qubit a;
      qubit b;
      qubit c;

      h a;
      x b-;
      hspace {2cm} -;
      x a;
      z b-;
      h a;
      z b-;
   \end{yquant}
\end{tikzpicture}
```

The vertical distance between registers is calculated by yquant automatically based on the height and depth that yquant find for this particular register—i.e., how much space is required above and below the wire line for all the gates. In order to enlarge these values, reset /yquant/register/minimum height or /yquant/register/minimum depth to a different value. It is not possible to artificially *reduce* the calculated heights and depths, as this would result in overlapping gates. However, sometimes it might be required to exclude a certain gate from the calculation; then, use the overlay attribute.

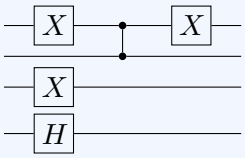

```
\begin{tikzpicture}
   \begin{yquant*}
      [x radius=1cm]
      x a;
      box {\hbox to 1cm{\hfil$X$\hfil}} a;
      hspace {1cm} a;
      x a;
      discard a;
   \end{yquant*}
\end{tikzpicture}
```

Here, we demonstrate two possibilities to enlarge a box: The first is by specifying its size in terms of the x radius or y radius keys beforehand. Those values serve as minimum sizes and would be extended if the text extended beyond the box. The second option is to just enlarge the text artificially by explicitly putting it into a fixed-width box. Note that in the first case, the *radius* is specified, i.e., the half-width, while in the second case, it is the *total* width (both times modulo the inner separation). Also note that the /yquant/operator/minimum width style is unsuitable for the given task: it would not change the visual width, only what yquant assumes its width to be.

B. Global Adjustment




```
\begin{tikzpicture}
   \begin{yquant*}[operator/separation=1cm]
      h a;
      phase {$\beta$} a;
      h a;
   \end{yquant*}
\end{tikzpicture}
```



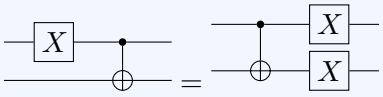

```
\begin{tikzpicture}
   \begin{yquant*}[register/minimum height=0pt,
   ↪  register/minimum depth=0pt]
      x a[0, 2];
      zz (a[0, 1]);
      x a[0];
      h b;
   \end{yquant*}
\end{tikzpicture}
```

Updated in 0.4, 0.1.1

By default, yquant will use the height and depth that is required by the individual gates, but at least /yquant/register/minimum height or, respectively, /yquant/register/minimum depth (which default to 1.5mm). Only manually reducing the default height will produce the cramped spacing displayed here.

C. Alignment

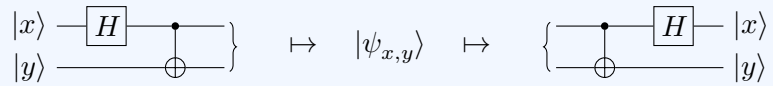
Updated in 0.5, 0.4



```
\begin{tikzpicture}
   \begin{yquant*}
      x a[0];
      cnot a[1] | a[0];
   \end{yquant*}
\end{tikzpicture}
$=$
\begin{tikzpicture}
   \begin{yquant*}
      cnot a[1] | a[0];
      x a;
   \end{yquant*}
\end{tikzpicture}
```

Not specifying anything for the vertical alignment will lead to the common TikZ problem: the baseline will be at the bottom, which is particularly bad in this case due to the missing X gate. The keys for minimal register sizes do not help here, since they only affect yquant's internal handling, but not the bounding box calculated by TikZ. The recommended way to draw circuit equations is always with the groups language extension.

```
% \useyquantlanguage{groups}
\begin{yquantgroup}
   \registers{
      qubit {} q[2];
   }
   \circuit{
      x q[0];
      cnot q[1] | q[0];
   }
   \equals
   \circuit{
      cnot q[1] | q[0];
      x q;
   }
\end{yquantgroup}
```



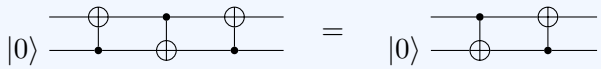

```
% \useyquantlanguage{groups}
\begin{yquantgroup}
   \registers{
      qubit {} q[2];
   }
   \circuit{
      init {$\ket x$} q[0];
      init {$\ket y$} q[1];

      h q[0];
      cnot q[1] | q[0];
      output {} (-);
   }
   \equals[$\mapsto\quad\ket{\psi_{x, y}}\quad\mapsto$]
   \circuit{
      init {} (q);
      cnot q[1] | q[0];
      h q[0];

      output {$\ket x$} q[0];
      output {$\ket y$} q[1];
   }
\end{yquantgroup}
```

Here, we do not have a circuit equation (i.e., logical statements involving multiple rather independent circuits), but a circuit progression. Since only in one circuit we have a description of the registers, we declare them without an initial text and put their initialization into `init` gates. The mapping in between is done by using the optional argument of the `\equals` macro. In order to obtain the braces at the ends, we use empty `output` and `init` gates.

Updated in 0.5
New in 0.4

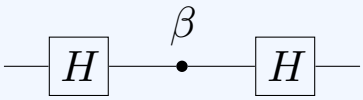
1. Perfecting Vertical Alignment

Updated in 0.5, 0.4




```
% \useyquantlanguage{groups}
\begin{yquantgroup}
   \registers{
      qubit {} q;
      qubit {$\ket0$} q[+1];
   }
   \circuit{
      cnot q[0] | q[1];
      cnot q[1] | q[0];
      cnot q[0] | q[1];
   }
   \equals
   \circuit{
      cnot q[1] | q[0];
      cnot q[0] | q[1];
   }
\end{yquantgroup}
```

D. Scaling

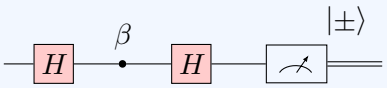



```
\begin{tikzpicture}[scale=1.5]
   \begin{yquant*}
      h a;
      phase {$\beta$} a;
      h a;
   \end{yquant*}
\end{tikzpicture}
```

Here, we first scaled the circuit itself. The default style for `/yquant/every circuit` sets the `transform shape` key for every node (which means any gate), so that those are also scaled. If your TikZ version is at least 3.1.6a, this is all that needs to be done. In earlier versions, there was a bug that required `yquant` to reset the `transform shape` key for `labels`, which would then require you to scale those manually.

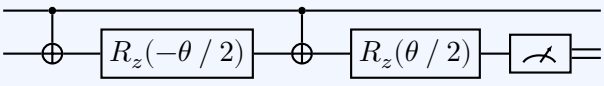
6.3.6 VII. Typesetting

A. Global Styling



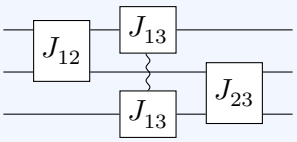

```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}[operators/every h/.append style={fill=red!20}]
      h a;
      phase {$\beta$} a;
      h a;
      ["$\ket\pm$" above right]
      measure a;
   \end{yquant*}
\end{tikzpicture}
```

Instead of setting `/yquant/operators/every h`, we could also have changed `/yquant/operators/every box`. Had we used `/yquant/every operator`, then the measurement would also have changed. Again, due to a TikZ limitation, it is not possible to change the position of labels on a per-style basis, only by using `label` options or a global setting.




```
\begin{tikzpicture}[thick]
   \begin{yquant*}[every operator/.prefix style={fill=white}]
      cnot a[1] | a[0];
      box {$R_z(-\theta\fracslash2)$} a[1];
      cnot a[1] | a[0];
      box {$R_z(\theta\fracslash2)$} a[1];
      measure a[1];
   \end{yquant*}
\end{tikzpicture}
```

As the “thin” style is the default, we present the opposite. By default, all operators are transparent; we changed this by giving all of them a white background color (but as a style *prefix*, so that, e.g., black fillings overwrite this). Contrary to quantikz, this also fills the cnots. If you only want to fill certain operators, you have to selectively target them using their styles.

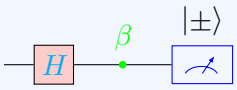



```
\begin{tikzpicture}
   \begin{yquant}[operators/every box/.append style={fill=white}]
      qubit {} j[3];
      box {$J_{12}$} (-j[1]);
      box {$J_{13}$} (j[0, 2]);
      box {$J_{23}$} (j[1]-);
   \end{yquant}
\end{tikzpicture}
```

`yquant` properly splits discontiguous multi-qubit operations.

Updated in 0.1.2, 0.1.1

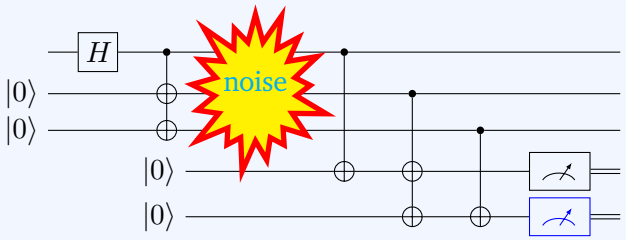
B. Per-Gate Styling




```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}
      [fill=red!20, font=\color{cyan}]
      h a;
      [green]
      phase {[green]$\beta$} a;
      ["$\ket\pm$", blue]
      measure a;
      discard a;
   \end{yquant*}
\end{tikzpicture}
```

Note that assigning styles in this way will forward them to `/yquant/operator style`, i.e., if you have controls, the style will not apply to them. `/yquant/style` is suitable to style both, e.g., `[style={fill=red!20}]`.

Updated in 0.4

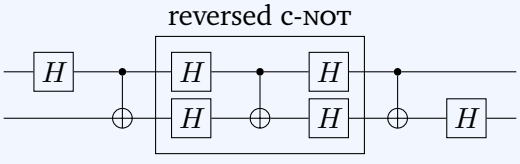


Updated in 0.6, 0.4

```
% \usetikzlibrary{shapes.symbols, fit}
\begin{tikzpicture}
   \begin{yquant}
      qubit {} data;
      qubit {$\ket0$} anc1[2];

      h data;
      cnot anc1 | data;
      [after=data]
      qubit {$\ket0$} anc2[2];
      [name=noise]
      text {\phantom{noise}} (data, anc1);
      cnot anc2[0] | data;
      cnot anc2 | anc1[0];
      cnot anc2[1] | anc1[1];
      measure anc2[0];
      [blue] measure anc2[1];
   \end{yquant}
   \node[starburst, cyan, fill=yellow, draw=red, line width=2pt,
         inner xsep=-4pt, inner ysep=-5pt, fit=(noise)] {noise};
\end{tikzpicture}
```

TikZ shapes cannot simply be used with yquant. Any yquant shape must be aware of the keys `x radius` and `y radius` that control its width and height. Additionally, yquant shapes must implement clipping paths. Those objects, which are a yquant addition to TikZ allow yquant to properly clip wires and vertical lines to the shape of the gate. yquant draws its elements sequentially; hence, a wire that comes into an operator will be hidden by anything the operator draws on top of it; but outgoing wires will in turn draw on the operator (modulo clipping). To avoid the issues, we construct an invisible box operator and name it; *outside* of the `yquant` environment, we `fit` the special TikZ shape on top of it.

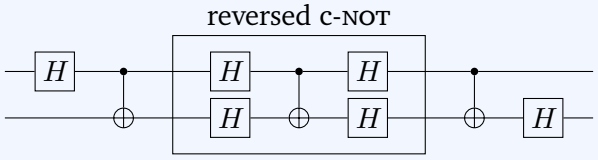
C. Boxing/Highlighting Parts of a Circuit



Updated in 0.2

```
% \usetikzlibrary{quotes, fit}
\begin{tikzpicture}
   \begin{yquant*}
      h a;
      cnot b | a;
      [name=left]
      h -;
      cnot b | a;
      [name=right]
      h -;
      cnot b|a;
      h b;
   \end{yquant*}
   \node[fit=(left-0) (left-1) (right-0) (right-1),
         draw, inner sep=6pt, "reversed c-\textsc{not}"] {};
\end{tikzpicture}
```

As usual, subcircuits provide a similar experience, but respect the separation:

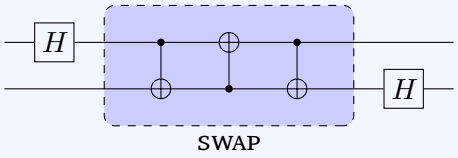



```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}
      h a;
      cnot b | a;
      [this subcircuit box style={inner ysep=6pt, "reversed
      ↪  c-\textsc{not}"}]
      subcircuit {
         qubit {} x;
         qubit {} y;
         h -;
         cnot y | x;
         h -;
      } (-);
      cnot b | a;
      h b;
   \end{yquant*}
\end{tikzpicture}
```

Here, we used the key `/yquant/this subcircuit box style` to influence only the style of the subcircuit box itself instead of providing global options

that apply to every object in the subcircuit (you wouldn't want the label be assigned to every single gate).

Updated in 0.4



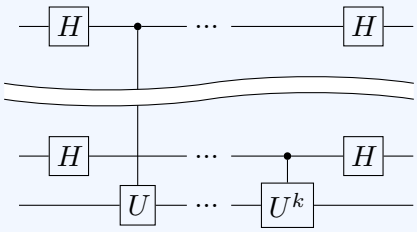

```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant*}
      h a;
      [this subcircuit box style={draw, dashed, rounded corners,
      ↪  fill=blue!20, inner ysep=10pt, "\textsc{swap}" below},
      ↪  register/default name=]
      subcircuit {
         qubit a;
         qubit b;
         cnot b | a;
         cnot a | b;
         cnot b | a;
      } (a-b);
      h b;
   \end{yquant*}
\end{tikzpicture}
```

Since version 0.2, fully enclosing a bunch of operations (with no controls extending to some inner component) is possible by means of `subcircuit`s. Before, this had to be done using named operations and layers. Note that here we used the style `/yquant/this subcircuit box style` to assign a styling that only applies to the box containing the subcircuit, but not to the inner gates—which would have happened had we just given the arguments to the subcircuit directly.

`yquant` does not support the fancy nearest-neighbor swap gate that `quantikz` has. It would however not be very difficult to implement this particular shape and make it available. Maybe even a multi-swap gate using the `knots` library would be possible.

6.3.7 VIII. Otherwise undocumented features

Updated in 0.6, 0.4

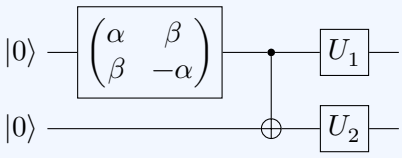



```
% \usetikzlibrary{quantikz,fit}
\begin{tikzpicture}
   \begin{yquant}[register/default name=]
      qubit a;
      [name=wave, register/minimum height=5mm, register/minimum depth=5mm]
      nobit wave;
      qubit b;
      qubit c;

      h a, b;
      box {$U$} c | a;
      text {$\dots$} a, b-;
      box {$U^k$} c | b;
      h a, b;
   \end{yquant}
   \node[wave, fit=(wave) (current bounding box.east |- wave), inner
    ↪  ysep=.5pt, inner xsep=0pt] {};
\end{tikzpicture}
```

Here, we included quantikz, which provides the `wave` shape, then introduced a register that will contain this wave (and enlarged it sufficiently). After the circuit is drawn, we `fit` the wave along. Since the name assigned to a register without any text actually is of a `coordinate` shape, we need to enlarge the height of the wave by providing a slightly increased `inner ysep`. Additionally, quantikz sets a negative `inner xsep`, which is probably required for its grid layout; but yquant positions exactly, so we also need to reset this.

yquant does not provide a shape corresponding to the “creating an ebit” gate.

6.3.8 X. Troubleshooting

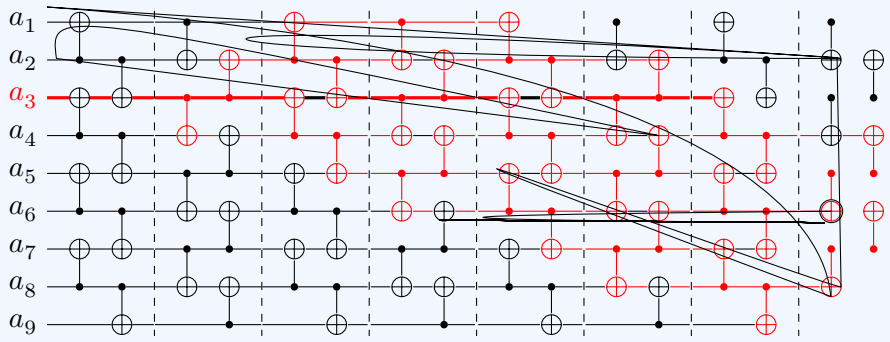



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {$\ket0$} a[2];
      box {$\begin{pmatrix}
               \alpha & \beta \\
               \beta & -\alpha
            \end{pmatrix}$} a[0];
      cnot a[1] | a[0];
      box {$U_{\The\numexpr\idx+1}$} a;
   \end{yquant}
\end{tikzpicture}
```

6.4 qpic documentation

Again, our section headings will be the same as in the qpic manual. As the qpic manual uses a bit larger separation between the operators than yquant's default, we globally say `\yquantset{operator/separation=3mm, register/default name=$\reg$, register/default lazy name=$\reg$}`, which allows to easily generate all the registers on-the-fly.

6.4.1 1 Introduction



```
\def\reversecircuit#1{%
   \begin{tikzpicture}
      \let\high=\empty
      \listeadd\high{#1}
      \def\cnot##1|##2;{%
         \ifinlist{##2}\high{
            \yquant [style=red] cnot a[##1] | a[##2];
            \ifinlist{##1}\high{
               \listremove\high{##1}
               \yquant addstyle {black} a[##1];
            }{
               \listadd\high{##1}
               \yquant addstyle {red} a[##1];
            }
         }{
            \yquant cnot a[##1] | a[##2];
         }
      }
      \def\cnotA{\cnot 0|1; \cnot 2|3; \cnot 4|5; \cnot 6|7;}
      \def\cnotB{\cnot 2|1; \cnot 4|3; \cnot 6|5; \cnot 8|7;}
      \def\cnotC{\cnot 1|0; \cnot 3|2; \cnot 5|4; \cnot 7|6;}
      \def\cnotD{\cnot 1|2; \cnot 3|4; \cnot 5|6; \cnot 7|8;}
      \def\cnotBlock{\cnotA \cnotB barrier (-); \cnotC \cnotD}
      \begin{yquant}[operator/minimum width=0pt, register/minimum height=2mm,
                     register/minimum depth=2mm]
         qubit {\Ifnum\idx=#1\color{red}\Fi$\reg_{\The\numexpr\idx+1}$} a[9];
         addstyle {very thick, red} a[#1];

         \cnotBlock barrier (-);
         \cnotBlock barrier (-);
         \cnotBlock barrier (-);
         \cnotBlock barrier (-);
         \cnotBlock
         output {\protect\xifinlist{\idx}{\high}{\color{red}}\relax
                 $a_{\The\numexpr9-\idx}$} -;
      \end{yquant}
   \end{tikzpicture}%
}
\reversecircuit2
```

This is an extremely interesting example, which could have been implemented in a lot of different manners. We chose an approach where we deferred the logic of coloring the gates entirely to TEX. Note that we put everything, including the whole `tikzpicture` itself, in a macro `\reversecircuit`, which we call directly after its definition by saying `\reversecircuit2`. This is of course an overkill in this situation—there is no need for the macro definition. However, note

that the macro expects the wire that is to be colored in red as its argument. So by slightly changing the invocation to

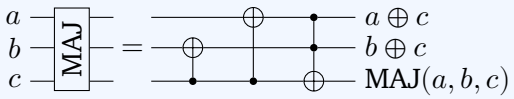
```
\foreach \ici in {0, ..., 8} {
   \reversecircuit\ici
   \par\vspace{1cm}
}
```

we are able to render the circuit with all different initial wires one after the other very easily. We do not show the output in the manual to keep it succinct, but just try it out by yourself.
We now explain what is done in the macro.
We first define an `etoolbox` list that is stored in `\high`. The idea is that this list holds at any point in time the indices of all the registers that are currently colored in red. Initially, we add the index that was given as a parameter to the macro—in our case, this was 2. Note we use `\listeadd` instead of `\listadd`, which is important for the invocation via `\foreach`—we want to have the number in the list, not the macro `\ici` that holds the index of the initially colored wire).
Next, we do not want to manually do the bookkeeping of this list. All we want to do is to issue the command to put a `cnot` gate on the appropriate registers and TEX should keep track of the correct coloring and register state. For this, we first define a macro `\cnot` that expects the index of the target and the index of the control register. Within this macro, we check whether the register of the control is currently highlighted (`\ifinlist{#2}\high`). If this is not the case, we draw the `cnot` gate without any additional styles (note that since we interrupted the `yquant` parser due to the lots of intervening macros, we first have to restart it saying `\yquant`). However, if it is the case that the control register is currently highlighted, we draw the `cnot` gate with the attribute `[style=red]`—we do not only want to draw the gate itself in red (for which `[red]` would be sufficient), but also its control line and the control blob, so we use the `/yquant/style` shorthand. Note that in the `qpic` manual, some of the control lines are thicker than others. This could be implemented by adding the argument `every control line/.append style={very thick}`; however, as it is unclear what the thick line should indicate, we did not add this to the example.
Then, we have to change the state of the target register appropriately, since the highlighting state will propagate from control to target. If the target register was already highlighted, we have to remove it from the list and we change its line color back to `black`; if it was not highlighted, we add it to the list and change its line color to `red`. Note that the use of `addstyle` will keep adding styles, so in the end, the register line style will be a long string `red,black,red,black,...`; we could do better by saying `setstyle`, which would overwrite the line style. However, since we will initially set the line width of `a[#1]` to `very thick`—which should be kept throughout the circuit—we would have to take additional care not to lose this setting. Here, we chose the simpler version.
After setting the coloring preliminaries, we note that if we slice the circuit at any time, it will

have four possible gate configurations (or a `barrier`). We define abbreviations for these in the macros `\cnotA` to `\cnotD`. Since they will always follow in this order, we also define a `\cnotBlock` abbreviation that executes these configurations together with their intermediate `barrier`. We do not include the final barrier, since it is not present in the last block.
Now, we start the `yquant` environment. We give some options for a nicer spacing and initialize the registers. In the `qpic` example, the registers are 1-indexed, so we use `\The\numexpr\idx+1` to give back the value of the current register index (`\idx`) plus one; we also conditionally color the register name in red if the index coincides with the parameter. Note that here we use `yquant`'s shorthand for `\protect\the`, `\protect\ifnum`, and `\protect\fi`, which are `\The`, `\Ifnum`, and `\Fi`, to get the correct expansion behavior.
Then, we add the initial style for the `a[#1]` wire. Inserting the gates together with the correct coloring is now extremely simple: we just need to call our `\cnotBlock` command and intersperse it with `barrier`s. At the end, we output all the gates in reverse order, which works similarly to the initialization of the gates, and also conditionally color the reversed register. This coloring could in principle be done similarly to the coloring of the initial label, saying something like `\Ifnum\numexpr8-\idx=#1 \color{red}\Fi`. Here, we chose the "more honest" approach to color all the registers that are still present in the coloring list—note the need to `\protect` the `etoolbox` macro `\xifinlist`. In this way, we could, e.g., terminate the circuit earlier and still get the correct output coloring at this particular point.
Note that if there were more that just five blocks, we could also have made use of Ti*k*Z's `\foreach` loop to output all the `\cnotBlock` commands. However, be aware of the fact that `\foreach` puts its content in a group, so the `\high` list assignments would have been local and forgotten in the next iteration. Either they would need to be made globally or some non-grouping loop construct would have to be used (e.g., `\pgfplotsforeachungrouped`).

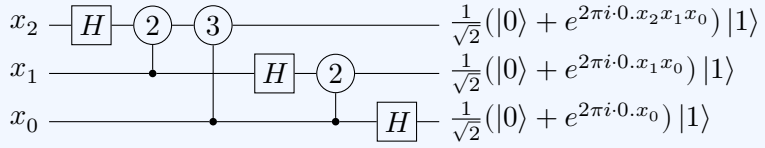
6.4.2 2 Simple Examples

2.1 Example 1: Majority




```
\begin{tikzpicture}
   \begin{yquant*}
      box {\rotatebox{90}{$\operatorname{MAJ}$}} (a, b, c);

      text {$=$} (-);

      cnot b | c;
      cnot a | c;
      cnot c | a, b;

      output {$a \oplus c$} a;
      output {$b \oplus c$} b;
      output {$\operatorname{MAJ}(a, b, c)$} c;
   \end{yquant*}
\end{tikzpicture}
```

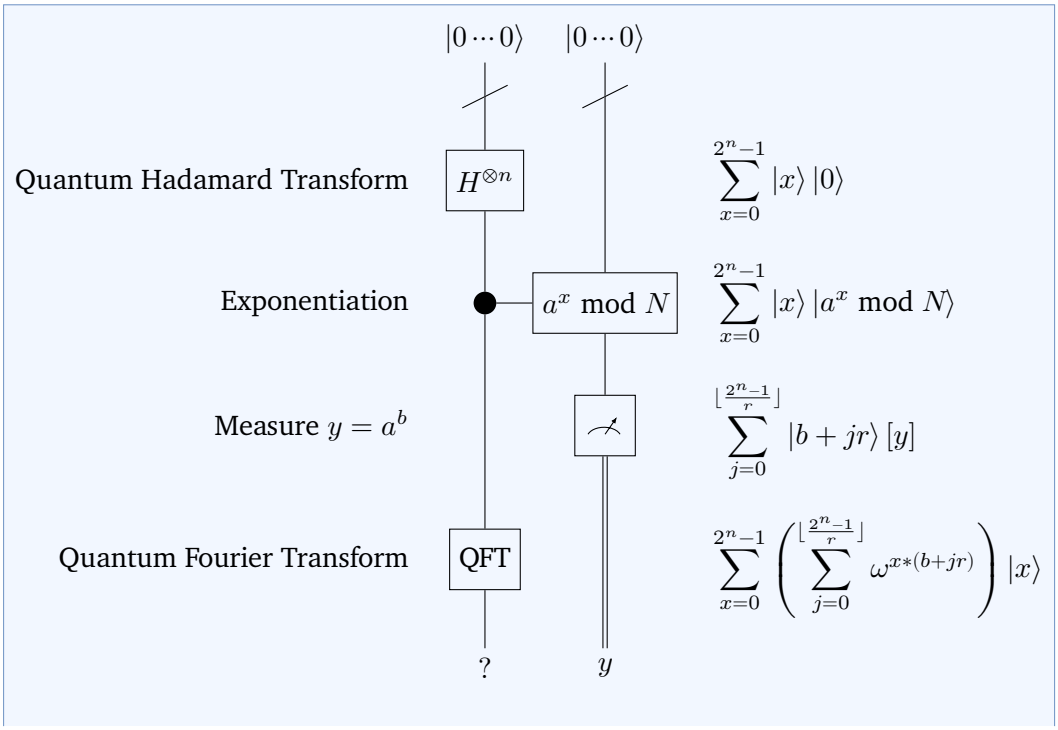
2.2 Example 2: Quantum Fourier Transform




```
\begin{tikzpicture}
   \begin{yquant}[operators/every box/.append style={shape=yquant-circle,
    ↪  radius=2.5mm}]
      qubit {$x_2$} x2;
      qubit {$x_1$} x1;
      qubit {$x_0$} x0;

      h x2;
      box {$2$} x2 | x1;
      box {$3$} x2 | x0;
      h x1;
      box {$2$} x1 | x0;
      h x0;

      output {$\frac{1}{\sqrt2} (\ket0 + e^{2\pi i \cdot 0.x_2 x_1 x_0})
       ↪  \ket1$} x2;
      output {$\frac{1}{\sqrt2} (\ket0 + e^{2\pi i \cdot 0.x_1 x_0})
       ↪  \ket1$} x1;
      output {$\frac{1}{\sqrt2} (\ket0 + e^{2\pi i \cdot 0.x_0}) \ket1$}
       ↪  x0;
   \end{yquant}
\end{tikzpicture}
```

In this example, we opted to use three distinct registers instead of one vector register, since the reversed indexing would probably have led to more confusion. We globally overwrite the `/yquant/operators/every box` style to use a circular shape instead. Note that this would usually be an ellipse, so we explicitly set the radius to a value that exceeds the minimum (half) width. As of version 0.6, the style `/yquant/operators/every box` is no longer the base style for other rectangular boxes such as `h` (note this requires a compatibility version of at least `0.6` or higher). For earlier versions, the change in shape would also affect the Hadamard gate and therefore would have to be reverted using the `/yquant/operators/every h` style.

2.3 Example 3: Shor's Algorithm



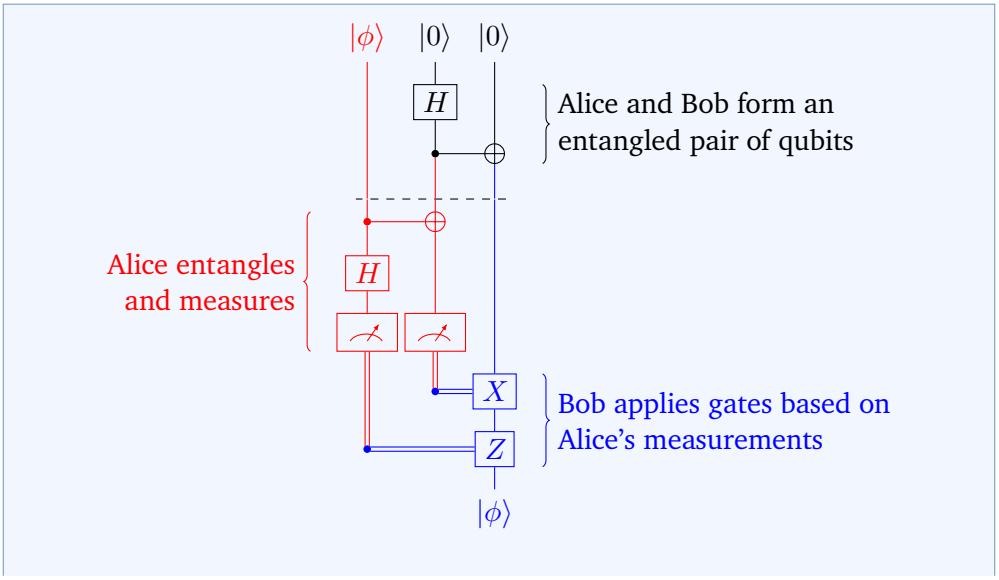
Updated in 0.7

```
\begin{tikzpicture}
   \def\explain#1#2{%
      \yquant
      [anchor=east, overlay=right] text {#1} explainLeft;
      [anchor=west, overlay=left] text {$\displaystyle#2$} explainRight;
      align -;
   }
   \begin{yquant}[vertical,
                  every control/.append style={radius=1.5mm},
                  operators/every slash/.append style={x radius=3mm, y
                   ↪  radius=1.5mm},
                  operators/every box/.append style={y radius=4mm},
                  operators/every measure/.append style={y radius=4mm},
                  operator/minimum extent=1.32cm]
      nobit explainLeft;
      qubit {$\ket{0\dotsm0}$} x; slash x;
      qubit {$\ket{0\dotsm0}$} y; slash y;
      nobit explainRight;
      align -;

      box {$H^{\otimes n}$} x;
      \explain{Quantum Hadamard Transform}
              {\sum_{x = 0}^{2^n -1} \ket x \ket 0}

      box {$a^x \bmod N$} y | x;
      \explain{Exponentiation}
              {\sum_{x = 0}^{2^n -1} \ket x \ket{a^x \bmod N}}

      measure y; output {$y$} y;
      \explain{Measure $y = a^b$}
              {\sum_{j = 0}^{\lfloor\frac{2^n -1}{r}\rfloor} \ket{b + j r} [y]}

      box {QFT} x;
      \explain{Quantum Fourier Transform}
              {\sum_{x = 0}^{2^n -1} \left( \sum_{j = 0}^{\lfloor\frac{2^n
               ↪  -1}{r}\rfloor}
                                            \omega^{x * (b + j r)} \right) \ket
                                             ↪  x}

      output {$?$} x;
   \end{yquant}
\end{tikzpicture}
```

Once again, this is a very interesting example. The circuit itself is very standard (though it is the first one in vertical mode). In order to output the explanations, we define two invisible wires together with a macro that populates the corresponding

`text` gates on those wires. Note that usually, in a vertical layout, all gates would be center-aligned. In order to align the left explanations to the right and the right explanations to the left, we just redefine the `anchors` of the gates. However, since `yquant` expects the gates to connect at the `center` anchor and hence equally distributes the required space for the wire to its left and right, this would lead to a very large gap between the explanations and the circuit itself. Here, we don't need `yquant`'s wire extent calculations at all—the explanation wires are invisible and at the border of the circuit. Therefore, we just disable the calculations using the `overlay` attribute. After all the explanations are done, we `align` all registers (if the minimum extent was chosen to be exactly larger or equal to the actual vertical extent of every gate, this would not be necessary; but with two decimals, this is probably not precisely the case).
Note that the code example above would work exactly in this way in traditional LaTeX documents; however, this document is set with `unicode-math`. This leads to a problem when using `\bmod`, which has to be `\protect`ed—or, as we did here, just say `\robustify\bmod` somewhere before its usage and after loading all the packages.

2.4 Example 4: Teleportation



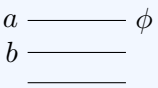
Updated in 0.7

```
\begin{tikzpicture}
   \newcommand\leftExplain[2][]{
      \yquantsecondpass{
         \draw[decoration={brace, mirror}, decorate, #1]
            let \p1=(leftComments), \p2=(explainTop-0.north),
             ↪  \p3=(explainBottom-0.south) in
            (\x1, \y2) -- (\x1, \y3) node[midway, left=2pt, align=right]
             ↪  {#2};
      }
   }
   \newcommand\rightExplain[2][]{
      \yquantsecondpass{
         \draw[decoration=brace, decorate, #1]
            let \p1=(rightComments), \p2=(explainTop-0.north),
             ↪  \p3=(explainBottom-0.south) in
            (\x1, \y2) -- (\x1, \y3) node[midway, right=2pt, align=left]
             ↪  {#2};
      }
   }
   \begin{yquant}[vertical]
      [name=leftComments] nobit explainLeft;
      qubit {\color{red}$\ket\phi$} q;
      qubit {$\ket0$} q[+2];
      [name=rightComments] nobit explainRight;
      setstyle {red} q[0];

      [name=explainTop]
      h q[1];
      [name=explainBottom]
      cnot q[2] | q[1];
      \rightExplain{Alice and Bob form an\\entangled pair of qubits}
      setstyle {red} q[1];
      setstyle {blue} q[2];
      barrier (q);

      [style=red, operator/separation=0pt, name=explainTop] cnot q[1] |
       ↪  q[0];
      [red] h q[0];
      [red, name=explainBottom] measure q[0, 1];
      \leftExplain[red]{Alice entangles\\and measures}

      [style=blue, name=explainTop] x q[2] | q[1];
      [style=blue, name=explainBottom] z q[2] | q[0];
      discard q[0, 1];
      \rightExplain[blue]{Bob applies gates based on\\Alice's
       ↪  measurements}

      output {\color{blue}$\ket\phi$} q[2];
   \end{yquant}
\end{tikzpicture}
```

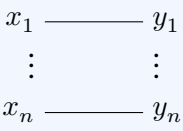
This example shows a different way of providing explanations along the circuit, which this time also works for hints encompassing more than a single gate. We define the macros `\leftExplain` and `\rightExplain`, which in the circuit will do the job. They rely on the existent of a couple of named nodes: The node `leftComments` (or `rightComments`) is the named node created at the beginning of the circuit when declaring the two invisible wires. They will serve with their horizontal position. Additionally, we require the named gates `explainTop` and `explainBottom`, which should correspond to the first and last gate that is supposed to be enclosed in the brace. We then use TikZ's `let` functionality in order to extract the required coordinates and draw the nodes. Note that the macros `\p`, `\x`, and `\y` come into existence only when the `\draw` command is executed; hence, just writing the `\draw` call would lead to an error: `yquant` would first try to add (using `\protected@edef`) the content of the macro to its output routine for the second pass, then call it in the first pass. We don't need the execution in the first pass—though it would not do harm—but we cannot allow for the expansion at this stage. Hence, we wrap the whole command in `\yquantsecondpass` (which automatically restarts the parser afterwards, so we don't have to do this).

6.4.3 3.1 Wires

3.1.1 Wire Declarations




```
\begin{tikzpicture}
   \begin{yquant}
      qubit a;
      qubit b;
      qubit {} c;

      hspace {1cm} -;

      output {$\phi$} a;
   \end{yquant}
\end{tikzpicture}
```

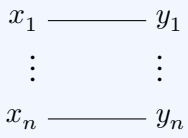



```
\makeatletter
% https://tex.stackexchange.com/a/112212/32357
\DeclareRobustCommand\rvdots{%
   \vbox{%
      \baselineskip4\p@\lineskiplimit\z@%
      \kern-\p@%
      \hbox{.}\hbox{.}\hbox{.}%
   }%
}
\begin{tikzpicture}
   \begin{yquant}[every nobit output/.style={},
    ↪  register/separation=3mm]
      qubit {$x_1$} x;
      qubit {$\rvdots$} x[+1]; discard x[1];
      qubit {$x_n$} x[+1];

      hspace {1cm} -;

      output {$y_1$} x[0];
      output {$\rvdots$} x[1];
      output {$y_n$} x[2];
   \end{yquant}
\end{tikzpicture}
```

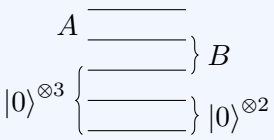
This is one of at least four possible implementations (note we defined `\rvdots`, since the native `\vdots` does not appear to be very well-centered). It declares the "invisible" register as part of the vector register `x`. As a consequence, whenever the whole vector register is addressed in operations, the operation is also drawn on the invisible register. For multi-register gates, this may be desired (they just span the whole region), for single-register gates, this is most certainly undesired. Note that `yquant` does not allow to declare an `output` gate for invisible registers—usually, this does not make sense. However, this is not enforced as a hard constraint, but rather due to the fact that the style `/yquant/every nobit output` does not exist. To prevent an error message, we just define this as an empty style. Also note that, to get a proper vertical spacing, we decided to use the `/yquant/register/separation` key—which works well, as there are no other registers. If there were others, it would be better to increase the height and depth of the invisible register.

Alternatively, we might declare the invisible register with a completely different name. This would create a discontiguous vector register `x`, which is probably the better thing to do for single-register gates. However, `yquant` may now try to split multi-register gates into contiguous slices—there could be arbitrary

registers between discontiguous parts of a vector register, and they should of course not be targeted if they are not in the list of targets of a gate—but here, we would actually want to have this.
A third approach mixes yquant and TikZ code. We declare a vector register with size two, manually increase, say, the depth of the first register, and put a \node at the appropriate position by naming the initial labels. In this way, vector usage will never target the "invisible" line—since it does not exist; both single- and multi-register gates will work appropriately. As a drawback, we need to decide whether we want to enlarge the depth of the first or the height of the second register (or both, splitting in half)—but what if some gates will actually be so large that they would provide enough of height or depth had we just chosen a different way of distribution the space?
In order to remediate this, a fourth way using only two registers would be to defer the drawing of the dots to a multi-register text gate which receives a y radius that enforces an appropriate separation. yquant will then automatically perfectly distribute the vertical extents among height and depth of the involved registers. The drawback with this approach is of course that the dots will be drawn *within* the circuit, not to the left. There is an undocumented option that we can use to shift the gate to the left; but since this only works for initializers, we still need to draw the dots for the outputs manually. A possible implementation could look as follows.




```
% \rvdots definition from above
\begin{tikzpicture}
   \begin{yquant}
      qubit {$x_1$} x;
      qubit {$x_n$} x[+1];
      [internal/move label, anchor=east, y radius=8mm]
      text {$\rvdots$} (x);

      hspace {1cm} -;

      [name=o1]
      output {$y_1$} x[0];
      [name=o2]
      output {$y_n$} x[1];
   \end{yquant}
   \path (o1.south west) -- (o2.north west) node[midway,
    ↪ /yquant/every output] {$\rvdots$};
\end{tikzpicture}
```



```
\begin{tikzpicture}
   \begin{yquant*}
      [decorate=false, draw=none]
      init {$A$} (q[0, 1]);
      init {$\ket0^{\otimes3}$} (q[2-4]);

      hspace {1cm} -;

      output {$B$} (q[1, 2]);
      output {$\ket0^{\otimes2}$} (q[3, 4]);
   \end{yquant*}
\end{tikzpicture}
```

Note that here the A is drawn without the curly braces. There are two simple ways to achieve this: by setting `draw` to `none`, the curly brace is suppressed, but still the A would be drawn at the same position as if the brace were there. We additionally set `decorate` to `false` to fully remove any reminiscence of the brace, so that the text is closest to the wires. (Note that just removing the decoration without also removing the drawing would lead to a vertical line that connects all the affected wires—this is how the `yquant-init` shape looks like in an undecorated fashion.)

6.4.4 3.2 Gates

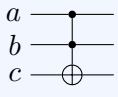
3.2.1 Controlled NOT and controlled Z

```
\begin{tikzpicture}
   \begin{yquant*}
      not a;
   \end{yquant*}
\end{tikzpicture}
```

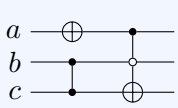
```
\begin{tikzpicture}
   \begin{yquant*}[register/default lazy
    ↪  name=$\symbol{\numexpr`a+\idx}$]
      cnot q[1] | q[0];
   \end{yquant*}
\end{tikzpicture}
```

Since in yquant's notation, `b` is mentioned before `a`, it would also be created as the first wire. If we instead resort to vector registers, we can directly specify

which position our registers should have. Of course, for longer circuits, an explicit declaration is probably favorable.

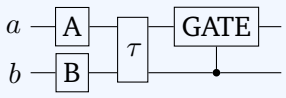



```
\begin{tikzpicture}
   \begin{yquant*}[register/default lazy
    ↪  name=$\symbol{\numexpr`a+\idx}$]
      cnot q[2] | q[-1];
   \end{yquant*}
\end{tikzpicture}
```



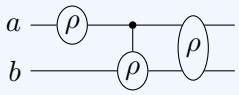

```
\begin{tikzpicture}
   \begin{yquant*}
      not a;
      zz (b, c);
      cnot c | a ~ b;
   \end{yquant*}
\end{tikzpicture}
```

3.2.2 General Gates



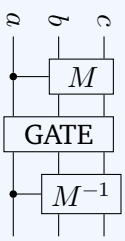

```
\begin{tikzpicture}
   \begin{yquant*}
      box {\symbol{\numexpr`A+\idx}} a, b;
      box {$\tau$} (-);
      box {GATE} a | b;
   \end{yquant*}
\end{tikzpicture}
```

Note that the macro `\idx` is available in any gate, and it gives the index of the current register within the target list.



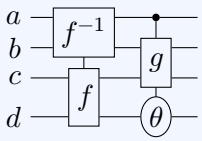

```
\begin{tikzpicture}
   \begin{yquant*}[operators/every box/.append
   ↪  style={shape=yquant-circle}]
      box {$\rho$} a;
      box {$\rho$} b | a;
      box {$\rho$} (-);
   \end{yquant*}
\end{tikzpicture}
```

Note that the macro `\idx` is available in any gate, and it gives the index of the current register within the target list.




```
\begin{tikzpicture}
   \begin{yquant}[vertical=-90]
      qubit a; qubit b; qubit c;
      box {$M$} (c, b) | a;
      box {GATE} (-);
      box {$M^{-1}$} (c, b) | a;
   \end{yquant}
\end{tikzpicture}
```

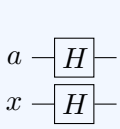
Here, we use the automatic rotation feature that the `/yquant/vertical` style provides.



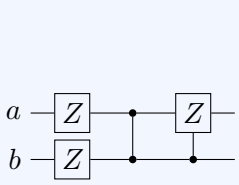

```
\begin{tikzpicture}
   \begin{yquant*}
      [name=fs] box {\Ifnum\idx<1 $f^{-1}$\Else $f$\Fi}
      ↪  (a, b), (c, d);
      [name=g] box {$g$} (b, c) | a;
      [shape=yquant-circle, name=theta] box {$\theta$}
      ↪  d;
   \end{yquant*}
   \draw (fs-0) -- (fs-1) (g) -- (theta);
\end{tikzpicture}
```

By putting the two f-boxes into a single gate, we ensured that `yquant` will center them with respect to each other.

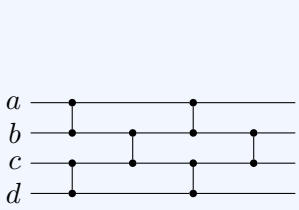
3.2.3 Other predefined Gates



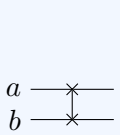

```
\begin{tikzpicture}
   \begin{yquant*}
      h a;
      h x;
   \end{yquant*}
\end{tikzpicture}
```



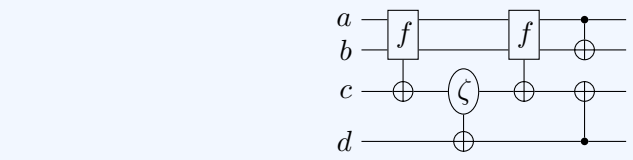

```
\begin{tikzpicture}
   \begin{yquant*}
      z a, b;
      zz (-);
      z a | b;
   \end{yquant*}
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant*}
      zz (a, b), (c, d);
      zz (b, c);
      zz (-b), (c-);
      zz (b, c);
   \end{yquant*}
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant*}
      swap (a, b);
   \end{yquant*}
\end{tikzpicture}
```



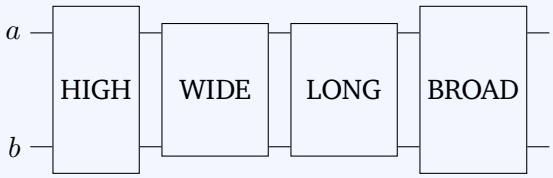
```
\begin{tikzpicture}
   \begin{yquant*}[plusctrl/.style={/yquant/every
   ↪  control/.style={/yquant/operators/every not}, /yquant/every
   ↪  positive control/.style={}}]
      [plusctrl] box {$f$} (a, b) | c;
      [plusctrl, shape=yquant-circle] box {$\zeta$} c | d;
      [plusctrl] box {$f$} (a, b) | c;
      cnot b | a;
      cnot c | d;
   \end{yquant*}
\end{tikzpicture}
```

This very unorthodox-looking style can be achieved by altering the control styles in such a way that it basically looks like a `not` gate.

6.4.5 3.3 Attributes

Size Attributes

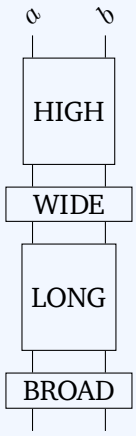
Changed in 0.7



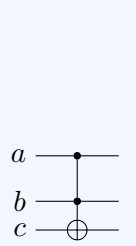

```
\begin{tikzpicture}
   \begin{yquant*}
      [y radius=20pt] box {HIGH} (a, b);
      [x radius=20pt] box {WIDE} (-);
      [time radius=20pt] box {LONG} (-);
      [space radius=20pt] box {BROAD} (-);
   \end{yquant*}
\end{tikzpicture}
```

In horizontal mode, `time radius` is a synonym for `x radius`, while `space radius` is a synonym for `y radius`.

Changed in 0.7

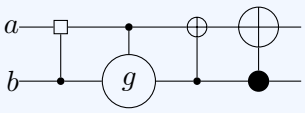



```
\begin{tikzpicture}
   \begin{yquant*}[vertical=45]
      [y radius=20pt] box {HIGH} (a, b);
      [x radius=20pt] box {WIDE} (-);
      [time radius=20pt] box {LONG} (-);
      [space radius=20pt] box {BROAD} (-);
   \end{yquant*}
\end{tikzpicture}
```



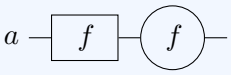

```
\begin{tikzpicture}
   \begin{yquant}
      [register/minimum depth=10pt]
      qubit a;
      qubit b;
      [register/minimum height=1pt]
      qubit c;

      cnot c | a, b;
   \end{yquant}
\end{tikzpicture}
```

Upon creation, the minimum register sizes can be passed on to `yquant`; note that the `/yquant/register/minimum height` extends from the wire line to the top of the space that is allocated for the wire, whereas the corresponding key `/yquant/register/minimum depth` extends from the wire line to the bottom. Hence, the values given here are half of `qpic`'s.



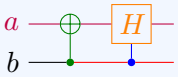

```
\begin{tikzpicture}
   \begin{yquant*}
      [inner sep=0pt, radius=2.5pt]
      box {} a | b;
      [shape=yquant-circle, radius=10pt]
      box {$g$} b | a;
      cnot a | b;
      [operator style={radius=7.5pt}, control
      ↪ style={radius=4pt}]
      cnot a | b;
   \end{yquant*}
\end{tikzpicture}
```

To mimick closely qpic's manual, we used an empty box instead of the xx gate, which also is a rectangle. Note that the shapes that accept text also have an inner separation, which would interfere with the radius setting.

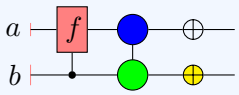



```
\begin{tikzpicture}
   \begin{yquant*}
      [x radius=12.5pt]
      box {$f$} a;
      [radius=12.pt, shape=yquant-circle]
      box {$f$} a;
   \end{yquant*}
\end{tikzpicture}
```

Note that when changing to the yquant-circle shape, this will become an ellipse if only one of the radii is modified.

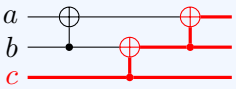



```
\begin{tikzpicture}
   \begin{yquant}
      qubit {\color{purple}$a$} a;
      setstyle {purple} a;
      qubit b;

      [style=green!50!black]
      cnot a | b;
      setstyle {red} b;
      [orange, control style=blue]
      h a | b;
   \end{yquant}
\end{tikzpicture}
```




```
\yquantdefinebox{circle}[shape=yquant-circle, draw,
↪  inner sep=0pt, radius=2mm]{}
\begin{tikzpicture}
   \begin{yquant*}
      [fill=red!50!white] box {$f$} a | b;
      [fill=blue, name=b] circle a;
      [fill=green, name=g] circle b;
      not a;
      [fill=yellow] not b;
   \end{yquant*}
   \draw (b) -- (g);
\end{tikzpicture}
```


```
\begin{tikzpicture}
   \begin{yquant*}
      setstyle {dotted} a;
      setstyle {very thick} b;
      [dashed, fill=yellow] box {$G$} (-);
      cnot a | b;
      setstyle {densely dotted} a;
   \end{yquant*}
\end{tikzpicture}
```



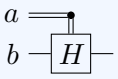

```
\begin{tikzpicture}[on/.style={red, very thick}]
   \begin{yquant*}[on/.style={style=red, control
   ↪  style={very thick}}]
      cnot a | b;
      qubit {\color{red}$c$} c;
      setstyle {on} c;
      [on] cnot b | c;
      setstyle {on} b;
      [on] cnot a | b;
      setstyle {on} a;
   \end{yquant*}
\end{tikzpicture}
```

This example demonstrates for the first time that new registers can also be declared at any later time in the circuit. Note that we defined two very different styles:

- `/tikz/on` for the `tikzpicture`

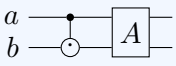
This is an ordinary TikZ style and hence will be applied whenever it is used in a styling context—for example, when added to the wire styles.

- `/yquant/on` for the `yquant*` environment
  This is a style that does not directly apply any styling, but it instead passes options to `/yquant/style` (we want to have the gates as well as their controls and control lines in red) as well as to `/yquant/control style` (we want to draw the control lines thicker [in principle, this would also affect the controls, but they are filled, not drawn], but we don't want to draw the lines of the `cnot` gates themselves be drawn thicker.). Since attributes for gates will first look in the `/yquant` namespace, this style is applied when used as an attribute for a gate (but beware that `[style=on]` would call the other style).




```
\begin{tikzpicture}
   \begin{yquant}[register/default name=$\reg$]
      cbit a;
      qubit b;
      h b | a;
      discard a;
   \end{yquant}
\end{tikzpicture}
```

yquant does not offer the variety of shapes that qpic does; please file a feature request if there is a need.

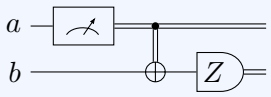



```
\begin{tikzpicture}
   \begin{yquant}
      qubit a;
      qubit b;
      [shape=yquant-circle, radius=1.2mm, inner sep=0pt]
      box {$\cdot$} b | a;
      box {$A$} (-);
   \end{yquant}
\end{tikzpicture}
```


```
% \usepackage[hidelinks]{hyperref}
% \usetikzlibrary{calc}
\makeatletter
\def\tikz@pgf@lib@fit@and@hull#1{%
   \ifdefined\tikz@alias%
      \unless\ifyquantmeasuring%
         \pgfqkeysalso{/tikz}{%
            % https://tex.stackexchange.com/a/36111/32357
            alias=sourcenode,
            append after command={
               % we don't need to worry about outer sep, yquant shapes
               ↪ ignore this value
               let \p1=(sourcenode.north west),
```

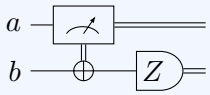
option only works within nodes). We also want to refrain from unnecessarily adding hyperlinks during the initial measurement phase. We then provide two TikZ styles to do the job, depending on whether the link should be created via **\hyperlink** or **\hyperref** and apply it. Note that here, we explicitly created the registers first. Had we used an implicit creation, we would also have applied to hyperref to the register labels!
Be aware of the fact that hyperlinks in PDFs will always be rectangular; if your gate shape is different from this, do not expect the shapes to match.

6.4.6 3.4 Measurement and Other Wire Type Changes




```
\begin{tikzpicture}
   \begin{yquant*}
      measure a;
      cnot b | a;
      dmeter {$Z$} b;
   \end{yquant*}
\end{tikzpicture}
```

yquant does not support the tag shape.

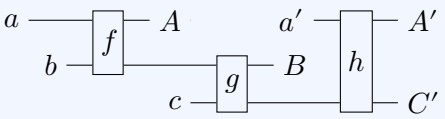



```
\begin{tikzpicture}
   \begin{yquant*}
      [direct control] measure a;
      cnot b | a;
      dmeter {$Z$} b;
   \end{yquant*}
\end{tikzpicture}
```

$a \dashv 0 \qquad 1 \vdash$

```
\begin{tikzpicture}
   \begin{yquant*}[operator/separation=2mm]
      setstyle {-|, shorten >= 3mm} a;
      inspect {$0$} a;
      discard a;

      hspace {1cm} -;

      setstyle {|-, shorten <= 2mm} a;
      init {$1$} a;
   \end{yquant*}
\end{tikzpicture}
```

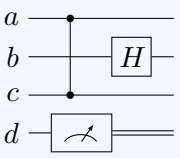
There are no gates in `yquant` that resemble the visual `discard` or reinitialization marker; however, this can be achieved by placing appropriate arrowheads at the wires. Still, this is a problematic solution: Every wire in `yquant` will extend from the center of one gate to the center of the next gate; protruding parts will be clipped away. Hence, the arrowhead will not be visible, as it is below the `inspect` or `init` gate—so we must `shorten` the wire by an "appropriate" amount. Additionally, if the circuit were longer, we would want to quickly get rid of this arrowhead style. `yquant` will try to make the wire lines as long as possible—i.e., in a normal circuit without any changes, the wire will in fact be one continuous line from the left to the right. However, whenever something changes at the wire—say, the style or type is changed—`yquant` needs to start a new path. We don't want the arrowheads to still be installed on this new path, hence we would quickly need to remove them.



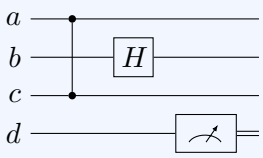

```
\begin{tikzpicture}
   \begin{yquant}
      qubit a;
      hspace {5mm} a;
      [after=a] qubit b;
      box {$f$} (a, b);
      inspect {$A$} a;
      [after=a] qubit c;
      discard a;
      box {$g$} (b, c);
      align -;
      init {$a'$} a;
      inspect {$B$} b;
      discard b;
      box {$h$} (-);
      output {$A'$} a;
      output {$C'$} c;
   \end{yquant}
\end{tikzpicture}
```

Here, we create registers with the `after` attribute at some later point in the circuit. Note that logically speaking, the h box should have had the targets `(a, c)`; however, as `yquant` does not know that the middle register was already discarded, it would have drawn two boxes joined by a wiggly line to indicate the discontiguous multi-qubit register.

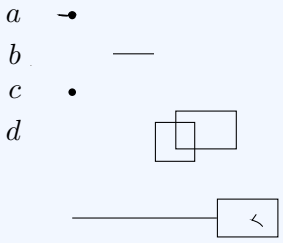
6.4.7 3.5 Managing Slices



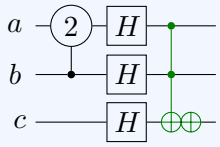

```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b; qubit c; qubit d;
      zz (a, c);
      h b;
      measure d;
   \end{yquant}
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b; qubit c; qubit d;
      zz (a, c);
      h b;
      align -;
      measure d;
   \end{yquant}
\end{tikzpicture}
```

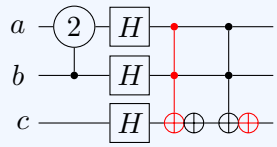


just an ordinary gate with a dashed line style; by the fact that all registers that are listed in the target list are aligned automatically, the `barrier` usually does its job. Hence, we need to `align` before the first `barrier`, as it does not perform an alignment by itself on registers that were not mentioned as targets. Here, we also change the default style (which is a dashed line) to the zigzag line that uses `qpic`'s style. Note that `yquant` automatically loads the library `decorations.pathmorphing`, so we don't need to do this.

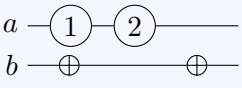



```
\begin{tikzpicture}
   \begin{yquant*}[operator/minimum width=0pt,
    ↪  operator/separation=2mm]
      [shape=yquant-circle, radius=1.5ex]
      box {$2$} a | b;
      h a, b, c;
      [style=green!50!black]
      cnot c | a, b;
      [operator/separation=0pt, green!50!black]
      not c;
   \end{yquant*}
\end{tikzpicture}
```

In order to stick two operators directly next to each other, we must set the `/yquant/operator/separation` to zero; this is the whitespace that is inserted before an operator. However, if the total with of an operator is smaller than `/yquant/operator/minimum width`, it is centered in a box of this width (giving a more uniform layout with lots of small gates), which would add additional whitespace both to the right of the Hadamards and to the left of the `cnot`s. We just globally suppress this minimum width, which is unproblematic for this particular circuit (we could also locally change it).



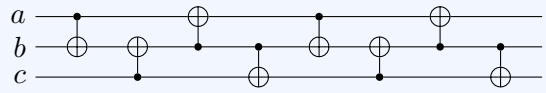

```
\begin{tikzpicture}
   \begin{yquant*}[operator/minimum width=0pt,
    ↪  operator/separation=2mm]
      [shape=yquant-circle, radius=1.5ex]
      box {$2$} a | b;
      h a, b, c;
      [style=red]
      cnot c | a, b;
      [operator/separation=0pt]
      not c;
      cnot c | a, b;
      [operator/separation=0pt, red]
      not c;
   \end{yquant*}
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant*}[operators/every box/.append
    ↪  style={shape=yquant-circle, radius=1.5ex}]
      box {$1$} a;
      not b;
      box {$2$} a;
      align -;
      not b;
   \end{yquant*}
\end{tikzpicture}
```

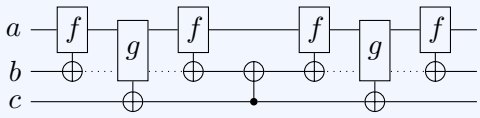
An instruction like `MIXGATES` does not exist in `yquant`, as it does not use a grid-based layout; but of course, its behavior can be faked by `align` gates.

6.4.8 3.6 Reversing and Repeating



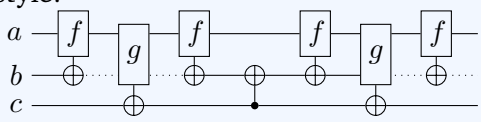

```
\begin{tikzpicture}
   \yquantdefinegate{cnots}{
      qubit a; qubit b; qubit c;
      cnot b | a;
      cnot b | c;
      cnot a | b;
      cnot c | b;
   }
   \begin{yquant*}
      cnots (a, b, c);
      cnots (-);
   \end{yquant*}
\end{tikzpicture}
```

yquant does not have a concept of slices and hence can also not automatically repeat gates within a certain slice. However, there are multiple ways to achieve the circuits in this section without repeating parts manually. Here, we defined a custom gate that contained the content and inserted it two times. Another alternative would be to do this using macros, as was illustrated in the very first example of the qpic section.

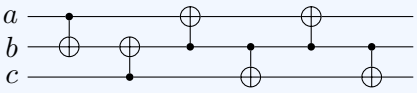


```
\begin{tikzpicture}
   \yquantdefinegate{gates}{
      qubit a; qubit b; qubit c;
      [plusctrl] box {$f$} a | b;
      addstyle {dotted} b;
      [plusctrl] box {$g$} (a, b) | c;
      [plusctrl] box {$f$} a | b;
      addstyle {solid} b;
   }
   \begin{yquant*}[plusctrl/.style={/yquant/every
    ↪  control/.style={/yquant/operators/every not}, /yquant/every
    ↪  positive control/.style={}}]
      gates (a, b, c);
      cnot b | c;
      gates (-);
   \end{yquant*}
\end{tikzpicture}
```

Note that for this sequence, we did basically the same thing as before, as we identified a symmetric slice—so reversing the order does not do anything.
We will also expand this example and give a very simple (and a bit shortsighted) implementation of a macro that reverses the order of gates. Note that this macro basically just splits its content at semicolons and when it is done inputs all the parts in reverse order. Hence, it will fail if semicolons appear, e.g., in attributes without enclosing them in braces. It also does not correspond exactly to the `R` instruction from `qpic`, as it does not *reverse* wire styles (basically `addstyle` would become a hypothetical `subtractstyle` macro), but just inserts them in reverse order. Hence, the following example will give a different circuit with respect to the wire style!



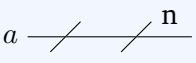
```
\makeatletter
\long\def\reversegates#1{%
   \begingroup%
      \let\reversegates@list=\empty%
      \count0=0 %
      \expandafter\reversegates@i#1;\reversegates@stop%
}
\long\def\reversegates@i#1;#2\reversegates@stop{%
   \ifstrempty{#2}{%
      \yquant@fordown \reversegates@idx := \count0 downto 1 {%
         \expandafter\expandafter\expandafter\yquant%
            \csname reversegates@list@\reversegates@idx\endcsname%
      }%
      \endgroup%
   }{%
      \ifstrequal{#2}{;}{%
         \reversegates@i;\reversegates@stop%
      }{%
         \advance\count0 by 1 %
         \csdef{reversegates@list@\the\count0}{#1;}%
         \reversegates@i#2\reversegates@stop%
      }%
   }
}
\begin{tikzpicture}
   \def\gates{%
      [plusctrl] box {$f$} a | b;
      addstyle {dotted} b;
      [plusctrl] box {$g$} (a, b) | c;
      [plusctrl] box {$f$} a | b;
      addstyle {solid} b;
   }
   \begin{yquant*}[plusctrl/.style={/yquant/every
    ↪  control/.style={/yquant/operators/every not}, /yquant/every
    ↪  positive control/.style={}}]
      \expandafter\yquant\gates
      cnot b | c;
      \reversegates\gates
   \end{yquant*}
\end{tikzpicture}
```



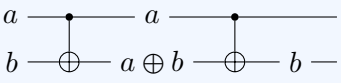

```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b; qubit c;
      cnot b | a;
      cnot b | c;
      \foreach \i in {1, 2} { \yquant
         cnot a | b;
         cnot c | b;
      }
   \end{yquant}
\end{tikzpicture}
```

Another way to repeat things is to just use appropriate repetition macros (and remember to restart the parser); here, we used `\foreach` from TikZ, but any other will also do the job.

6.4.9 3.7 Other Circuit Elements

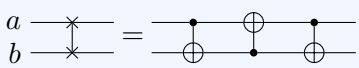



```
\begin{tikzpicture}
   \begin{yquant*}[operators/every slash/.append
    ↪  style={radius=2mm}]
      slash a;
      [label=10:n] slash a;
   \end{yquant*}
\end{tikzpicture}
```



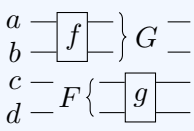

```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b;
      cnot b | a;
      inspect {\Ifcase\idx$a$\Else$a \oplus
       ↪  b$\Fi} -;
      cnot b | a;
      inspect {$b$} b;
   \end{yquant}
\end{tikzpicture}
```

Here, we achieved the centering of the two `inspect`ed registers by putting them in a single gate instruction with case discrimination.



```
\begin{tikzpicture}
   \begin{yquant*}
      swap (a, b);
      text {$=$} (-);
      cnot b | a;
      cnot a | b;
      cnot b | a;
   \end{yquant*}
\end{tikzpicture}
```

This is a very simple equality; for more complicated ones, the `groups` library is recommended.

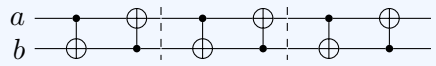


```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b; qubit c; qubit d;
      box {$f$} (a, b);
      inspect {$G$} (a, b);
      init {$F$} (c, d);
      box {$g$} (c, d);
   \end{yquant}
\end{tikzpicture}
```

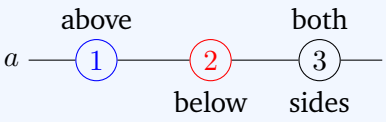
Note that we first defined all the registers explicitly, and they all use an initializing text. Had we directly used the `init` gate on the registers `(c, d)` as the *first* gate when *neither* of both registers had an initializing text, then the F would have been placed to the left of the wires. Basically, a `qubit` declaration with a value is the same as declaring the register without a value plus another `init` gate that puts the value in place. A zero-length `hspace` gate or an `align`ment directly at the beginning would be a way to prevent this shift to the left from happening.

`yquant` does not support the permutation gate that `qpic` has. It would however not be very difficult to implement this particular shape and make it available. Maybe even a multi-swap gate using the `knots` library would be possible.

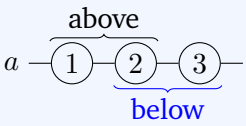
6.4.10 3.8 Comments



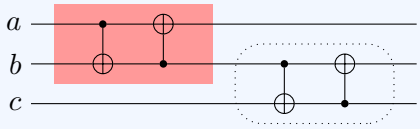

```
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b;
      \foreach \i in {0, 1, 2} { \yquant
         cnot b | a;
         cnot a | b;
         \ifnum\i<2 \yquant
            [operator/separation=2pt, operator/minimum width=0pt]
            barrier (-);
         \fi
      }
   \end{yquant}
\end{tikzpicture}
```




```
% \usetikzlibrary{quotes}
\begin{tikzpicture}
   \begin{yquant}[operators/every
   ↪  box/.append
   ↪  style={shape=yquant-circle,
   ↪  radius=1.5ex}]
      qubit a;
      [blue, "above" above] box {$1$} a;
      [red, "below" below] box {$2$} a;
      ["both" above, "sides" below] box
      ↪  {$3$} a;
   \end{yquant}
\end{tikzpicture}
```

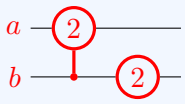


```
% \usetikzlibrary{calc}
\begin{tikzpicture}
   \begin{yquant}[operators/every box/.append style={shape=yquant-circle,
   ↪  radius=1.5ex}]
      qubit a;
      [name=1] box {$1$} a;
      [name=2] box {$2$} a;
      [name=3] box {$3$} a;
   \end{yquant}
   \draw[decoration=brace, decorate]
      ($(1.north west)+(-.1,.1)$) -- ($(2.north east)+(.1,.1)$)
      node[midway, above=1pt] {above};
   \draw[blue, decoration={brace, mirror}, decorate]
      ($(2.south west)+(-.1,-.1)$) -- ($(3.south east)+(.1,-.1)$)
      node[midway, below=1pt] {below};
\end{tikzpicture}
```




```
\begin{tikzpicture}
   \begin{yquant*}
      [this subcircuit box style={draw=none, fill=red!40!white}]
      subcircuit {
         qubit {} a; qubit {} b;
         cnot b | a;
         cnot a | b;
      } (a, b);
      [this subcircuit box style={draw, dotted, rounded corners=10pt}]
      subcircuit {
         qubit {} a; qubit {} b;
         cnot b | a;
         cnot a | b;
      } (b, c);
   \end{yquant*}
\end{tikzpicture}
```

6.4.11 3.9 Macros and LaTeX Code




```
\begin{tikzpicture}[loud/.style={red, very thick}]
   \yquantdefinebox{phase2}[loud, draw,
    ↪  shape=yquant-circle, radius=1.5ex]{$2$}
   \begin{yquant*}
      [style=loud]
      phase2 a | b;
      phase2 b;
   \end{yquant*}
\end{tikzpicture}
```

Note that here, we choose an alternative gate name, as there already is the built-in gate `phase`. While we could overwrite it, this is generally a very bad idea. Keep in mind that gate declarations are global and also that gates are case insensitive, so changing the capitalization would not help. Finally note that when we define a style for a new gate, it only pertains to the *gate* itself. It is not possible to change styles external to the gate—such as control lines—within the gate definition itself.

The following example in the `qpic` manual requires some additional thoughts. It defines a custom gate with a variable number of target registers. The `\yquantdefinegate` interface does not officially allow for this, although some low-level hacking can of course be done (a *sorted* `etoolbox` list is provided in the macro `\yquant@circuit@subcircuit@param`, which holds the internal indices of all currently involved target registers).
It is of course always possible to write some macros that output the required gate commands. Looking at the particular example, it is actually not really necessary to define a gate that has a variable number of targets. Rather, in `yquant`, one would define a new gate that just contains the two $\oplus$ symbols next to each other; the control line is drawn separately from the gate anyway. This very straightforward description will unfortunately fail, for the following reason: When a control line is drawn, `yquant` currently always draws it from the `center` anchor of the current shape upwards or downwards. However, for the $\oplus\!\!\oplus$ shape, the control line point should actually be in the middle of the right $\oplus$. This is an off-center point, so we need some hacking to convince `yquant` to do this. The following code is pretty long; we will therefore give parts of the code, followed by an explanation.
First of all, we define a shape (similar to what is done in `yquant-shapes.tex`) that holds the two $\oplus$es.

```
\makeatletter
\pgfdeclareshape{yquant-doubleoplus}{%
   \inheritsavedanchors[from=yquant-slash]%
   \anchor{center}{\pgfqpoint{.5
```

saved anchors and anchors (for details, see the TikZ manual, section 106.5.3, “Command for Declaring New Shapes”). The `x radius` now corresponds to the *diameter* of one of the circles, since we have two circles next to each other. We do not define border anchors at the moment; they would require some additional computation, but in lots of scenarios, they are not necessary (as we would also not need most of the anchors, but it is always good to have them). The clip path is also not very special, it just contains the shape that is to be clipped away; basically, both circles. The background path deserves more attention. We draw the two circles and the vertical and horizontal lines; but note that we disable the pgf's size protocol for all but the right circle. Hence, when this shape is used, TEX and yquant will actually think that it only occupies space for the right circle; the left one will protrude in the margin. (Actually, we could wrap this in a test such as `\ifdefined\yquant@prefix` to only discard protocoling within a `yquant` environment, so that the shape is properly usable outside.)
Next, we must take care of re-inserting this “lost” margin whenever the gate is used; and we also define a style that appropriately uses the shape:

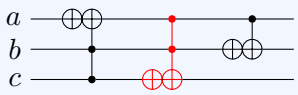
```
\yquantset{
   operators/every noffoli/.style={
      shape=yquant-doubleoplus, x radius=2.6mm, y radius=1.3mm, draw
   },
   internal/noffoli shift/.code={%
      \begingroup%
         \expandafter\tikzset\expandafter{\yquant@draw@@style}%
         \tikzset{/yquant/every operator, /yquant/operators/every noffoli,
          ↪  /yquant/this operator}%
         \edef\cmd{%
            \endgroup
            \dimdef\noexpand\yquant@config@operator@sep{%
               \yquant@config@operator@sep+
               \pgfkeysvalueof{/tikz/x radius}%
            }%
         }%
         \cmd
   }
}
```

The first style is almost a copy of the `/yquant/operators/every not` style, only with the new shape and a doubled `x radius`. The second style is more complicated: Protected in a group, it first sets all the custom style overwrites that a user may pass to the gate (e.g., the user may wish to overwrite the

radii)—those are stored in the internal macro `\yquant@draw@@style`. Then, it applies the styles in the order as the gate would do it; note that an attribute such as `[x radius=1cm]` would only add the directive to the `/yquant/this operator` style, but not apply it yet, therefore we now execute all the options that were stored previously. As the final action that survives the group, we add the current value for the `x radius` to the current value of the operator separation—this effectively enacts the proper placement of our gate.
Having defined those styles, we finally need to declare the gate itself, so that it can be used in a circuit:

```
\yquant@langhelper@declare@command
   {noffoli}
   {}
   {%
      \appto\yquant@attrs@remaining{,/yquant/internal/noffoli shift}%
      \yquant@prepare
         {}%
         {/yquant/operators/every noffoli}%
   }%
\yquant@langhelper@setup@attrs{noffoli}{}{}
```

We call `\yquant@langhelper@declare@command`, as for every standard gate declaration, with the desired name of the gate (`{noffoli}`), the actions that are to be carried out *before* the targets and controls are parsed (`{}`), and the actions that are to be carried out once the targets and controls are known. We append the style that we just defined to the list of attributes, and execute the gate preparation. Finally, we also declare the attributes that this gate takes—no required and no optional attributes.
After all this work, which can be saved in some shared document and used whenever necessary, we can come to the application, which is now very straightforward (however, note that our gate declaration was too simplistic for vertical mode; but an adaptation is not difficult).




```
\begin{tikzpicture}
   \begin{yquant*}
      noffoli a | b, c;
      [style=red] noffoli c | a, b;
      noffoli b | a;
   \end{yquant*}
\end{tikzpicture}
```


```
% \usetikzlibrary{backgrounds}
\begin{tikzpicture}
   \begin{yquant}
      qubit a; qubit b;
      [name=cn]
      cnot b | a;
      \draw[fill=blue] (cn) circle[radius=5pt];
      box {FONT} (-);
      [name=cn, fill=boxBlueBody]
      cnot b | a;
      \scoped[on background layer]
      ↪ \draw[fill=red] (cn)
      ↪ circle[radius=5pt];
   \end{yquant}
\end{tikzpicture}
```

There are no special options to mix TikZ code with yquant code, as this can be done natively at any time. To draw at the position of another gate, just name the gate. Note that—as was illustrated here—also using the same name multiple times is possible, in this case, the latter use overwrites the former. There is no direct equivalent to the PRETIKZ option—the gate has to be drawn first in order to get its position. However, TikZ supports layers, so it is easy to draw something behind a gate: just put it on a background layer, e.g., the one provided by the backgrounds library. Also note that here, we filled the second cnot gate with our background color in order to give the same image as in the qpic manual. The circle is not filled by default, hence the red "outer" circle would be visible also inside the cnot circle.

The HYPERTARGET instruction can be directly reproduced in TEX by just putting a \hypertarget before the tikzpicture.

7 Foreign language support and extensions

yquant is built in various modules, so that it is not hard to use the quantum circuit rendering backend, but expose a different language frontend. yquant not only understands its own language, but also others. Although we refer to "foreign languages," additional extension packages of the yquant language itself are also covered in this section and can be loaded by the same syntax.

7.1 groups

By saying \useyquantlanguage{groups} in the preamble after loading yquant itself, additional support for groups of yquant circuits is loaded. Various circuits in a group share a common set of registers, are appropriately aligned horizontally if on the same line and can also be aligned vertically among multiple lines. The main intended use is for circuit equations.

This extension provides the environment yquantgroup)i)L im{nse

7.1.2 Special macros

`\registers` The `\registers` macro can and must only be used once in a `yquantgroup` environment. It contains the declaration of all the registers that are shared among the various circuits within a group. Basically, if you follow the convention in a usual `yquant` circuit to first declare all the registers, then use the gates, then you would put the declaration part in the `\registers` macro. However, note that it is in principle also possible to mix register declarations with other gates and TikZ commands.

In case you do not use the `import` gate in any of the circuits within the group and you do not declare own registers, the behavior is very straightforward: basically, the content of `\registers` is copied verbatim at the beginning of each circuit. Otherwise, the general rule is: importing a register will ensure that all non-declaration commands that preceded this register declaration are executed; and importing the last register will additionally execute all succeeding commands within `\registers`.

`\circuit[<style>]{<content>}` The `\circuit` macro can be thought of as starting a `yquant` (or `yquant*`) environment and using its mandatory argument `<content>` as the content of the circuit; the optional `<style>` is used to apply additional styling options to the circuit.

This is not entirely accurate: In reality, the content is put into a `subcircuit` and `<style>` is passed as arguments to the subcircuit.

The default style `/yquant/operators/every group circuit` is applied to the circuit. This style is configured such that the illusion of working in a top-level `yquant` environment is very convincing: The circuit is frameless by default and uses the transparent name mangling scheme.

All the registers that were previously defined via `\registers` are automatically available within the circuit, as if their declaration had been copied. In fact, `yquant` will make a register available the first time it is referenced in some gate; if at the end of a circuit some of the shared registers were not used, they will be imported before exiting the circuit. Consequently, if you define own registers just for a single circuit, these will always be at the very top. This can be influenced by means of the `import` gate, which is only available in group `\circuit`s. This gate allows to import a declared register at an arbitrary position.

`\equals*[<content>]` The `\equals` macro inserts a blank text—internally, a `box`-like gate with the style `/yquant/operators/every group equals`—that contains `<content>`. If omitted, `<content>` is given by `$=$`.

The optional star will put a horizontal alignment mark at the position where the box is inserted. Similar to the `&` operation in `amsmath`'s `align` environment or the `\>` in TEX's native `tabbing`, `yquant` will now remember the horizontal position of the box internally and will allow you to directly jump to this position in the next line. Note that you may well have multiple alignment marks in a single line, which `yquant` internally numbers `1`, `2`, ….

`\\[<separation>]` The `\\` macro inserts a line break (never a page break), so that the next `\circuit` or `\equals` will be put below all circuits that were output before, and it will again start at the same left position as the first circuit. The default vertical distance is given by `/yquant/group/line separation`, but it may be overwritten by the optional `<separation>` argument, which must be a TEX dimension.

Note that if you *set* new alignment marks in a new line, this will delete the alignment marks that were previously set.

If the option `/yquant/group/aligned` is passed to the `yquantgroup` environment, the command `\shiftright` is implied after each linebreak.

`\shiftright*[<where>]` The `\shiftright` command will put the "cursor," i.e., the horizontal position at which the next `\circuit` or `\equals` will start, at the position specified by `<where>`. By default, `<where>` is `1`. If the optional star is present, `yquant` will additionally put an alignment mark at this position (see the documentation for `\equals`). If the option `/yquant/group/aligned` is passed to the `yquantgroup` environment, the command `\shiftright` is implied after each linebreak or starred page break.

The option `<where>` can take various forms:

- It may be a natural number `1`, `2`, …, denoting the number of an alignment mark specified in a previous line.
- It may be the number `0`, denoting the very beginning of the line; this is useful if the `/yquant/group/aligned` option is given, but for a specific line, no alignment should be performed.
- It may be a TEX dimension, in which case this dimension is directly added to the cursor (so it is a relative value). This is where passing the optional star makes most sense. If you want to position absolutely, you may first issue `\shiftright[0]` followed by a shift by the dimension that you want.

The macro is named \shiftright; however, yquant does not enforce that the actual position is to the right of the current position. You may indeed be able to create overlapping circuits if you shift back to a previous position.

\pagebreak*, \newpage*, \clearpage*, \cleardoublepage* The page breaking commands are available only if the yquantgroup was not enclosed in a tikzpicture. They will end the current picture environment, issue the original page breaking command, and start a new picture. Hence, if you want to pass options globally to the picture, you should use the /yquant/preamble option for the yquantgroup; the content of this key will be passed as options for every implicitly started tikzpicture.

Usually, remembering the horizontal alignment marks on a new page does not make much sense. For this reason, the commands will delete all alignment; use their starred versions to retain them. If the option /yquant/group/aligned is passed to the yquantgroup environment, the command \shiftright is implied after the starred version of the page break.

Typically, you will not want to refer to named gates in a circuit on a different page; remember that if you need this feature, you must pass the remember picture key in the /yquant/preamble option, as this is a reference to another tikzpicture. Also don't forget to use the (TikZ) overlay key on the corresponding path that references the node in order not to mess up with the bounding box (see the TikZ documentation for those two keys).

7.1.3 Configuration

Loading the groups language extension will define several new configuration keys.

/yquant/group/every group default:
Style that is installed for every yquantgroup and yquantgroup* environment, as if it had been given as an option. The style's default path is /tikz.

/yquant/group/line separation default: 5mm
This is the default vertical line separation that is inserted whenever a new line is issued in a yquantgroup.

/yquant/group/aligned default: false
This boolean flag defines whether \shiftright is automatically issued after \\ and the starred page breaking commands.

`/yquant/preamble` default:

This style may only be passed to the `yquantgroup` alignment directly as an option; it is not available via `\yquantset` and the like. It is only relevant if the `yquantgroup` is not contained in a `tikzpicture`. The content of this style will be given as an optional argument to the `tikzpicture`; this is the recommended way to specify TikZ options, as they are automatically preserved among page breaks.

`/yquant/operators/every group circuit` default: `/yquant/operators/every subcircuit, /yquant/operators/subcircuit/frameless, /yquant/operators/subcircuit/name mangling=transparent`

This style is installed for the `subcircuit` that implicitly wraps each `\circuit`. Note that some magic is carried out to ensure that the name mangling setting only applies to the *direct* content of the `\circuit`; any `subcircuit`s within the `\circuit` will use the default name mangling scheme.

`/yquant/operators/every group equals` default: `shape=yquant-rectangle, align=center, anchor/.expanded=\ifyquanthorz{center}{north west}, inner xsep=1mm, x radius=2mm, y radius=2.47mm` Changed in 0.7

This style is installed for every `\equals`, which is internally realized similarly to a `box` gate.

7.1.4 Gates and operations

No gates or operations may be used directly within the `yquantgroup` environment, but all the usual `yquant` gates and operations are available within `\registers` and `\circuit`. Additionally, within `\circuit`, the `import` gate is available.

`import`
Syntax: `import <target>;`
This is a pseudo-gate that makes all the outer registers given in `<target>` available in the current circuit. Consequently, the register names that are specified in `<target>`, also ranges, do not refer to the registers in the *current* `\circuit`, but instead to those defined via `\registers`. Therefore, it is for example possible to import all outer registers at once using `import -;`. Vector registers can also be imported partially.

If additional content (TEX commands such as TikZ paths, non-creation gates) is used within `\registers`, everything that comes *before* the declaration of a register will be copied into the `\circuit` when the register is imported; for a vector, this refers to the index zero. Additionally, any additional content that comes *after* the

declaration of the last register will be copied directly after the last register was imported.

Out-of-order importing

Note that it is principle possible to import registers out-of-order. Since matching outer and inner wires in subcircuits is done in the order in which they appear, this will lead to inner registers with names that do not match their outer registers and is probably highly undesirable.

Usually, this gate will not be needed as `yquant` will automatically import an outer register upon its first use.

Possible attributes: none

7.1.5 Vertical layout

New in 0.7

This library is aware of vertical circuits. Note that the layout *between* the circuits will always be the same, irrespective of the actual circuit orientation: Circuit will be set from left to right, and line breaks will always lead to a vertical shift and a reset of the horizontal position.

That said, the circuits may *internally* be set in vertical mode. Every invocation of `\circuit` will then always restart a new `yquant` environment. While the content will still be put in a lonely `subcircuit`—to keep consistency in the styling options—no inter-circuit wire alignment will be carried out.

There is an additional complication regarding the vertical alignment of circuit and equality signs within one "line." Now, the circuits can very well have a varying height, so vertically centering the circuits with respect to each other would not lead to a satisfying layout. We might desire to vertically center the equality sign between its two enclosing circuits. However, what if the line contains more than a two circuits and they all have different heights? Then, the equality signs would be at different positions. For this reason, all `\circuit`s and `\equals` will be aligned at their top.

7.2 qasm

By saying `\useyquantlanguage{qasm}` in the preamble after loading `yquant` itself, the parser for `qasm` (not OpenQASM) is loaded. It provides the environment `qasm` as well as the macro `\qasmimport`, which works similarly to `\yquantimport` (but does not accept additional options).

7.2.1 Language specification

The qasm language is not formally defined, but an overview is provided at the archived website of qasm2circ. The yquant implementation is designed to be compatible with the original parser, with the following exceptions:

- In qasm, lines could begin in an arbitrary manner; the first whitespace followed by the first valid command were then the instruction. Contrary to this, yquant's parser always expects a line to start with a valid gate (preceded by arbitrary whitespaces), a comment, or to be empty.
- In qasm, user-defined gates will be drawn in a box unless they contain the text \dmeter, and they will be recognized as measurement gates if they contain \meter or \dmeter. Contrary to this, yquant's parser expect the gates to *start* with one of the *macros* **\meter**, **\dmeter**, or **\dmeterwide**. Using these macros *within* the content of a gate does not make sense from the point of view that in yquant, gates are nodes with shapes, so either the full gate has a particular shape or it does not, but not only parts of it.
- The space gate is supposed to produce a horizontal whitespace without a gate. In yquant's implementation, you have to discard the wire if you want to reproduce this behavior; space and nop are equivalent.

The default qasm style defines several macros that can be used in gates. yquant makes **\m** (matrix; requires amsmath) and **\txt** (switch to text mode) available within the qasm environment.

Do not expect yquant's output to match the one of qasm exactly. yquant is not grid based, so that commands such as nop don't even make sense. They are implemented for compatibility reasons and will produce a fixed horizontal space of the operator minimum width plus one separation, which might or might not be accurate.

Note that whatever you write between **\begin**{qasm} and **\end**{qasm} is essentially treated as verbatim; only where the specification says so (in the definition of a new gate and in the optional third command to the register definition), it is interpreted as TEX markup. Consequently, in beamer, any frame containing these environments must be given the fragile option.

7.2.2 Configuration

Loading the qasm language interpreter will define several new configuration keys. For all the gates, it will use the keys defined in section 3, and it additionally provides the following:

/yquant/operators/every s default: /yquant/operators/every box
This style is installed for every s operator.

/yquant/operators/every t default: /yquant/operators/every box
This style is installed for every t operator.

/yquant/operators/every utwo default: /yquant/operators/every box
This style is installed for every Utwo operator.

/qasm/zero default: \qasm@ket0
The content of this macro is used as the initialization content whenever the zero gate is invoked.

/qasm/register/default qubit name default: \qasm@ket{#1}
This macro is invoked with a single parameter (the name of a qubit register) and gives back what is printed as the name of the register (will be in math mode automatically).

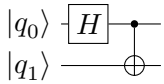
/qasm/register/default qubit name value default: \qasm@ket{#1} = \qasm@ket{#2}
This macro is invoked with two parameters (the name of a qubit register and its initial value) and gives back what is printed as the name of the register (will be in math mode automatically).

7.2.3 Examples

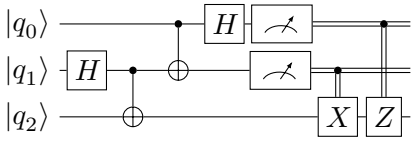
The unaltered .qasm files provided from the qasm2circ page were stored in the subfolder qasm relative to this manual's TEX file. The following command is then used to print all of them:

```
% preamble:
% \usepackage[compat=<version>]{yquant}
% \usepackage{import}
% \useyquantlanguage{qasm}
\def\yquantimportpath{qasm/}
\foreach \circuitno in {1, ..., 18} {
   \paragraph{Circuit \#\circuitno}
      \begin{center}
         \qasmimport{test\circuitno.qasm}
      \end{center}
}
```

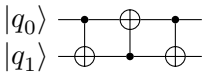
Circuit #1



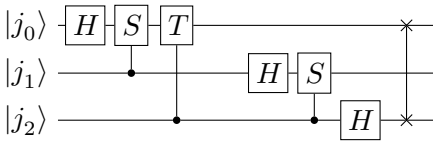
Circuit #2



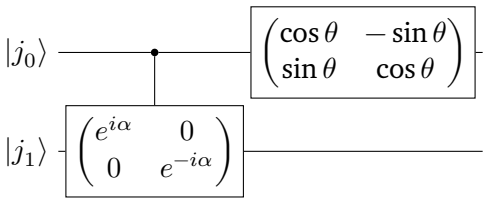
Circuit #3



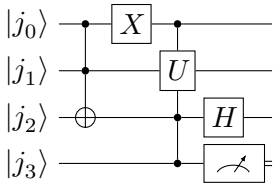
Circuit #4



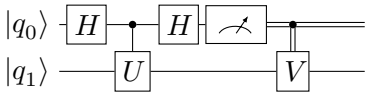
Circuit #5



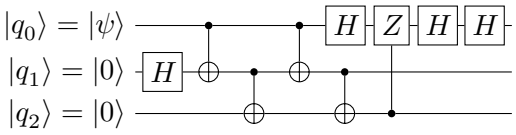
Circuit #6



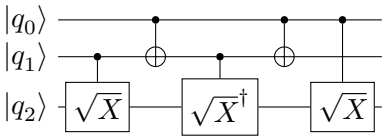
Circuit #7



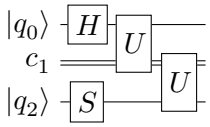
Circuit #8



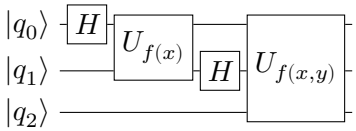
Circuit #9



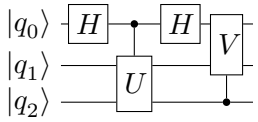
Circuit #10



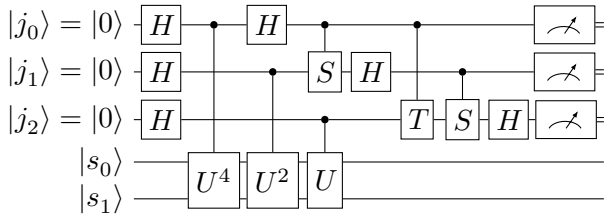
Circuit #11



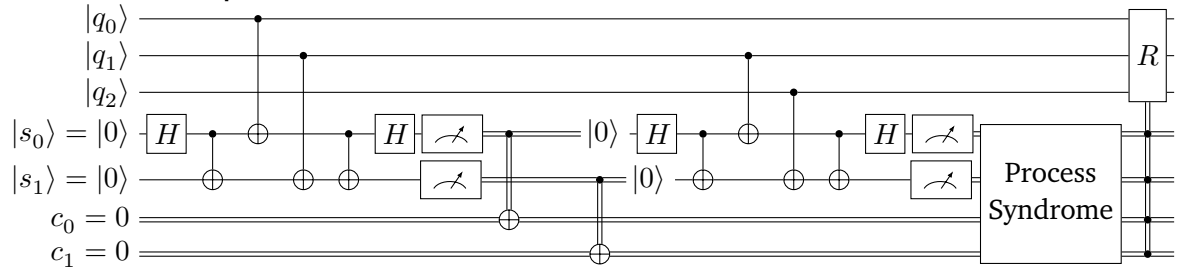
Circuit #12



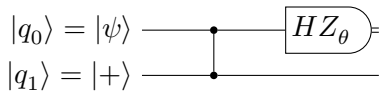
Circuit #13



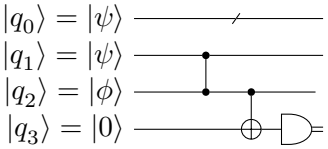
Circuit #14



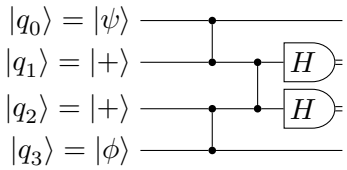
Circuit #15



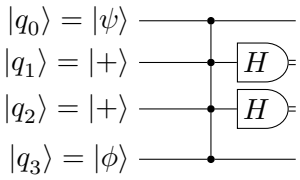
Circuit #16



Circuit #17



Circuit #18



8 Integration with other packages

In general, yquant should not introduce incompatibilities with other packages. However, the possibility to mix yquant code with arbitrary TEX code may lead to certain expectations on how things should work, which may not always be met. This is mainly due to the fact that yquant requires two passes of its content (see section 4.3), as it has to measure the heights and depths of the individual gates. Similar issues can for example also arise in amsmath's `align` environment, which also has a measuring and a shipout stage. If you run into an incompatibility using a macro from another package (or even plain TEX), try the following:

1. Using a TEX macro within yquant code will stop the yquant parser. Hence, all gates following this macro will be ignored. Did you remember to issue `\yquant` in order to restart the parser after your macro?
   Symptom: No errors, but gates are missing

2. Is the macro robust? Modern packages could automatically take care of this by a `\protected` definition, but older ones may not. Try to prefix the macro by `\protect`.
   Symptom: Unexpected error messages

3. Does the macro depend on other macros defined *within* the circuit? The double pass may lead to problems. If possible, define your macros outside of the `yquant` environment. If this is not possible, make sure the definitions are expandable, then at measurement stage, yquant will do the expansion, so that you get the correct results.
   Symptom: Only the latest assignment will show up whenever the macro is used; the vertical spacing (for subcircuits, possibly also the horizontal spacing) may even be screwed.

4. Does the macro create output, using its own font? If the output depends on the current position, this position will be completely wrong. This is due to the fact that the macro is executed at the first pass only, where yquant does not know about any positions at all. The macro `\yquantsecondpass` will defer its content so that it is executed only at the second pass, where positions are known. It will also automatically restart the parser. Note that `\yquantsecondpass` will not expand its content. If you need expansion, you may use `\yquantesecondpass`, which uses `\protected@edef`.
   Symptom: Content occurring in the wrong place or missing

   New in 0.6

5. Does the macro need to be executed at both passes? There may be reasons for this, in particular if you use commands from pgf's basic or system layer.

However, all custom macros will only be executed once, at the first pass. Wrap the macros in `\yquantescape` in order to execute them both times. The parser will automatically be restarted afterwards. Note that `\yquantescape` will not expand its content. You may use `\yquanteescape`, which will first expand its content using `\protected@edef`. Note that the content will first be executed, then stored for the second pass. New in 0.6
Symptom: Content missing

6. If all of this did not work and the use case is interesting enough, please file a bug report.

8.1 TikZ

yquant is built on top of TikZ an hence integrates well with TikZ. You can use all `\path`-like comands as well as scopes and yquant will automatically take care of restarting the parser appropriately. If it does not, this is very likely a bug, please file a report.

You should typically not use `\tikzset`, as all changes made by this macro will only be executed in the first pass. Instead, use `\yquantset` and change the path appropriately: this macro will first store its argument (using `\protected@edef`) for the second pass and then set the appropriate options also in the first pass.

Note that low-level pgf functions are not altered by yquant; this would be highly inefficient. If you need to use them, wrap them in `\yquantescape` or `\yquanteescape`. New in 0.6

8.2 beamer New in 0.6

yquant integrates with beamer overlays. This means that you can use the overlay commands `\only`, `\alt`, `\temporal`, `\uncover`, `\visible`, and `\invisible` directly in your yquant code; the parser will automatically be restarted whenever necessary. Note that the `*env` environments (`onlyenv`, `altenv`, `uncoverenv`, `visibleenv`, and `invisibleenv`) should *not* be used within yquant code.

The macros `\pause` and `\uncover` are also supported to some degree (`\uncover` with braces is fully supported). They should work well in simple circuits, but unexpected results can be expected in more complex scenarios. If you don't get appropriate results, use the aforementioned macros.

You may also use `\note` inside yquant circuits.

Note that yquant does not overwrite the definitions of the beamer macros, which implies that you can also use them *within* gates (e.g., for the `value` of a `box`). However, this means that the yquant parser must be running to detect these

macros and take appropriate action. Hence, if you interrupted the code via some special macros, make sure to restart the parser even if your next macro is, e.g., `\only`.

Finally note that `\only`, `\alt`, and `\temporal` are more special than the usual `yquant`-code-interrupting macros. They will *not* terminate the group that was opened for the current gate. As a consequence, you can also use these macros for arguments. Note that whenever you pass arguments to a gate using the `[<arguments>] <gate> <registers>;` syntax, the value of `<arguments>` is directly fed to `\pgfkeys`, which *does not* understand `beamer` macros. Hence, you *cannot* use, e.g., `\only` *within* the brackets. However, you can wrap the arguments including the brackets as a whole in `\only`—this happens before `yquant` relinquishes control to `\pgfkeys` and therefore is executed as expected. Since you can also pass multiple arguments to a gate by repeating `[<arguments>]`, this easily allows to combine arguments with and arguments without overlays.

Using overlays for arguments

```
% \documentclass{beamer}
\begin{frame}
   \begin{tikzpicture}
      \begin{yquant}
         qubit a;
         [fill=yellow]
         \alt<2>{[draw=blue]}{[draw=green]}
         \only<3>{[ultra thick]}
         h a;
      \end{yquant}
   \end{tikzpicture}
\end{frame}
```

The Hadamard gate will always be filled yellow; its line color will be blue on the second frame and green on all other frames. On the third frame, its line width is dramatically increased.

9 Changelog

2020-03-15: Version 0.1

Initial release

2020-03-22: Version 0.1.1

Complete rewrite of the register name parser. `yquant` now understands comma-separated lists and ranges in indices, and also is far more tolerant with respect to whitespaces.
`yquant` now also supports non-contiguous vector registers and allows to add new registers into an already existing vector that is not the last register, and also in the unstarred mode.

2020-04-11: Version 0.1.2

Introduce `setstyle` and `addstyle` pseudo-gates that allow to style individual wires; rename `setwire` to `settype` (the old name is still available and shows a deprecation warning).
Complete rewrite of the way `yquant` draws wires; projection anchors are removed in favor of clipping paths. This allows perfect connections between gates and wires, even if the (rather rectangular) wire lines meets with nonplanar shapes, while still preserving the possibility of transparent wires.
`yquant` now also properly draws non-contiguous multi-qubit operations.
New gate: `correlate`. Various bug fixes.

2020-06-02: Version 0.2

Introduce `subcircuit`; required rewriting how `yquant` internally positions vertically. Provide simple macros to load circuits (or parts) from a file and to declare own custom gates.

2020-06-07: Version 0.2.1

Introduce a macro to declare a lightweight custom gate, which is only a single box with custom content.

2020-06-13: Version 0.3

Introduce support for the `qasm` language.

2020-07-11: Version 0.3.1

Add legacy support for very old TikZ versions such as the one used on the arXiv.

2020-08-24: Version 0.3.2

Fix #5: Can't draw circuits with more than 9 qubits.

2020-10-27: Version 0.3.3

Fix #6: shorten doesn't work for 2-qubit barriers. This fixes a bug in how the `shorten` keyword worked on `barrier`s, which may require re-assessing your chosen values.

2021-02-21: Version 0.4-alpha

Lots of internal fixes, most notably vertical alignment with subcircuits.
Introduce capability to perform vertical alignment with multi-register gates.
Dramatic changes under the hood regarding horizontal positioning, which is now only determined in the drawing stage; this paves the way for delayed gates, which are planned for 0.4. Also changes in the gate declaration interface.
Introduce compatibility layer, so that layout-breaking changes will not become effective unless explicitly requested.
Separate register height into a height and depth key.
Introduce `overlay` key to disable height caluclation selectively.
Change register style declaration, so that this is now always equivalent to creating an unnamed register followed by an `init` gate with the given text. Note: This may be a **breaking change** that cannot be compatibility-protected—if you used TEX conditionals involving `\idx` for creation labels of registers, you will now need to either `\protect` them all or just capitalize their first letter (which corresponds to auto-`\protect`ed versions for `compat` at least `0.4`).
Now use nodes for `init` and `output` gates.
Change behavior of `hspace` and `align`: Now also extend if the wire is discarded afterwards.
Introduce the commands `\Ifnum`, `\Ifcase`, `\Or`, `\Else`, `\Fi`, `\Unless` and `\The` available for use within gates that behave like auto-`\protect`ed versions of their plain TEX equivalents.

2021-03-27: Version 0.4

New gate: `inspect`. Various bug fixes.
Introduce the `direct control` feature: `measure` gates can now substitute positive controls of future gates.

2021-07-03: Version 0.4.1

Fix #9: Output bracket misaligned.
Fix #10: Unable to access node in subcircuit. As of this version, named nodes in subcircuits will also be properly aliased if there is only a single target subcircuit (so that you don't need to use the `-0` suffix for the subcircuit's name).

2021-08-17: Version 0.5

Improvement: Active outer canvas transformations (TikZ shifts, scalings, rotations) should be supported more nicely (no guarantees!).
Improvement: Custom gates (`\yquantdefinegate`) can now contain TikZ `\path`-like commands without the `\noexpand` prefix.
Improvement: Automatically discard wires *inside* a subcircuit (even if they had the `out` or `inout` attribute) if they are discarded directly after the subcircuit *and* they have output gates within the subcircuit (else, the wire would be re-drawn from the output to the border of the subcircuit).
Bugfix: Referencing named gates in nested subcircuits now works without producing an error (worked before, but gave errors).
Bugfix: Properly handle the `direct control` feature if it was specified, but not used until the end of the (sub)circuit.
Introduce name mangling options for subcircuits.
Implement #11: Circuit equations. As of this version, the `groups` language is available that allows to easily implement circuit equations.

2021-09-04: Version 0.5.1

Bugfix: The `xx` gate style was not made available since version 0.4-alpha.

2021-12-28: Version 0.6

Bugfix: Support to set the `/yquant/operators/subcircuit/seamless` property outside of `yquant` environments.
New: Support for `beamer`.
New macros: `\yquantsecondpass`, `\yquantesecondpass`, `\yquantescape`, and

`\yquanteescape` for more fine-grained control of when to execute macros.
Bugfix: the auto-`\protect`ed versions `\Ifnum` etc. now also work in `output` gates.
New gate: `text`.
Introduce the `/yquant/operators/every rectangular box` style as a common ancestor of gates such as `box`, `h`, ... instead of using the `/yquant/operators/every box` style for this. As a consequence, boxes can now be styled globally without affecting the other gates. This new behavior is compatibility-protected.
Bugfix: `output`s can now be named.
Bugfix: Border angles of `yquant-ellipse` now work properly.
New configuration: `/yquant/register/default lazy name`.
Documentation: Replace the wishlist by a section on integration with other packages.
Documentation: Include the examples of `qpic` in this manual.

2022-02-05: Version 0.7

New: Support for vertical layout.
Various bugfixes.
Internal change of the loading order of the package files.
Fix #18: Subcircuit boxes don't render in `groups` environment. Now, styles can properly modify the `/yquant/operators/this subcircuit box` style without affecting the content of the subcircuit, only the box.
New: Support for a simplified drawing pipeline without clipping paths.

2022-05-07: Version 0.7.1

Fix issues with using handlers for attributes.
Fix #21: CSWAP scaling problems. Now the clipping of the `swap` and `slash` gate scale appropriately when scaling canvas transformations are in effect (no guarantee with rotations!). Multi-register lines in the same style as control lines (`/yquant/operator/multi as single`) will be automatically hidden if control lines are present—they would be drawn on top of each other in the exact same style.

2022-12-24: Version 0.7.2

Fix #23: Add the `iswap` gate.

2023-01-21: Version 0.7.3

Fix #24: Clipping in subcircuits can under very special circumstances fail. Change