$\langle\mathsf{q}|\mathsf{pic}\rangle$: Quantum Circuit Diagrams in LaTeX

Thomas G. Draper   Samuel A. Kutin

March 2016

Abstract

$\langle\mathsf{q}|\mathsf{pic}\rangle$ is a system for preparing circuit diagrams in LaTeX, with an emphasis on diagrams used in quantum computing. The user prepares a description of the circuit in the human-readable "$\langle\mathsf{q}|\mathsf{pic}\rangle$ language": a Python program then converts this into LaTeX code using the Ti*k*Z graphics package. This note serves as a manual for the language as of $\langle\mathsf{q}|\mathsf{pic}\rangle$ version 1.0.0 (March 2016).

Contents

1 Introduction

A picture is worth approximately a thousand words. For example, Figure 1 depicts a circuit [2] for reversing the information on n wires in depth $2n+2$, where the allowed operation is adding one wire to an adjacent wire. The proof that this circuit is correct takes a page and a half (and over 600 words), but the main idea of the proof is conveyed entirely by the red coloring in Figure 1.

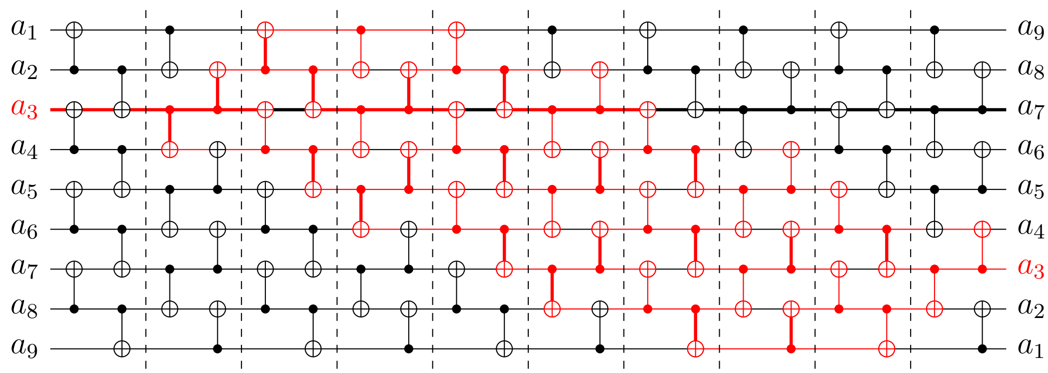


Figure 1: Reversing the contents of nine wires in depth 20 [2] using controlled NOTs. The red wires and gates are those affected by the value of a_3.

This diagram was created using ⟨q|pic⟩. The user creates a description of the circuit in ⟨q|pic⟩: a simple, human-readable language, with one line declaring each wire and one line specifying each logical operation, or "gate". A Python script, which we call[1] `qpic`, parses this description and produces LaTeX code using the Ti*k*Z graphics package.[2]

One design principle behind ⟨q|pic⟩ is to keep the language simple. The user can draw a number of elements common to quantum circuit diagrams, including wires with single or double lines, rectangular gates with text on them, and measurement boxes. The code is interpreted by LaTeX, so any math expressions are passed directly through. The goal is not to handle everything anyone might ever want; instead, commands can be passed directly through to LaTeX or to Ti*k*Z, so the user can supplement the ⟨q|pic⟩ output as needed. (`qpic` produces heavily commented Ti*k*Z code to facilitate this process.)

There are, of course, some limitations on what ⟨q|pic⟩ can achieve in this framework. The Python code that creates the Ti*k*Z commands needs to know how large all the elements are; so, for example, to make a box large enough to contain some text, the user must explicitly specify the width. Hence, it may take several iterations to line up the diagram perfectly, as is normal when constructing diagrams in LaTeX.

Another design principle is that, to the extent possible, the ⟨q|pic⟩ interpreter first creates an internal representation of the circuit from the user's input, and then produces Ti*k*Z code from that representation. One example of this is the `VERTICAL` command; with one line, the

[1]`qpic` stands for "⟨q|pic⟩ Python-interpreted compiler".
[2]The bulk of our testing has been with Ti*k*Z version 2.10.

user can change the default flow of time in the diagram (left to right) to a vertical flow (top to bottom). The internal representation of the circuit is the same in horizontal or vertical mode; only the output changes. In practice, the line between representation and code is not always as clear-cut as one would like.

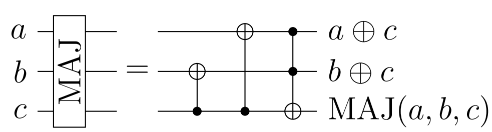
In this paper we give a brief, but complete, description of ⟨q|pic⟩. We begin with some simple examples in Section 2; the discussion should serve as a beginning tutorial. We give a complete list of commands of the ⟨q|pic⟩ circuit specification language in Section 3. We then briefly describe `qpic` in Sections 4 and 5, including instructions to help use ⟨q|pic⟩ within a LaTeX document. Finally, we complete our definition of the ⟨q|pic⟩ language by listing the gory details of parsing and tokenizing in Appendix A.

Like any software, ⟨q|pic⟩ is a work in progress. This paper represents a description of the state of the program as of March 2016. Please see `github` for the most up-to-date version of `qpic` and ⟨q|pic⟩. ⟨q|pic⟩ is copyright IDA/CCR-P, and is distributed under GNU General Public License v3. Please feel free to contact the authors with suggestions for further improvement.

2 Simple Examples

2.1 Example 1: Majority

We begin with a simple in-place majority circuit [1] in Figure 2. The figure includes the diagram and also the ⟨q|pic⟩ code used to generate it.




```
a W a {a \oplus c}
b W b {b \oplus c}
c W c \mbox{MAJ}(a,b,c)

a b c G \rotatebox{90}{MAJ}
=
+b c
+a c
a b +c
```

Figure 2: In-place majority vote: diagram and code.

The first three lines of the ⟨q|pic⟩ code use the `W` command, which declares a wire. The string before `W`, which is required, is the name of the wire, to be used internally. The formal rules for wire names can be found in Section 3.1.1—in brief, a wire name can be anything that doesn't mean something else to ⟨q|pic⟩—but any string of numbers and lowercase letters is safe. Here, the wires are given the names `a`, `b`, and `c`.

If a `W` command has an argument after the `W`, it is interpreted as a label to be typeset in math mode and placed at the start of the wire. A second argument (if any) is treated the same way but placed at the end. Here, each wire has a starting and ending label, representing its starting and ending value.

Normally, the ⟨q|pic⟩ parser breaks up each line by whitespace. However, an expression like `{a␣\oplus␣c}` is not broken up. ⟨q|pic⟩'s target language is LaTeX, so curly braces (and dollar signs) can be used to group text with whitespace into a single entity. See Appendix A for more about ⟨q|pic⟩'s parsing rules.

After a blank line (ignored by ⟨q|pic⟩), line 5 contains our first gate. The basic syntax of a gate line is a list of targets, the gate (possibly with a required argument), and then a list of controls (if any). In this case, the gate type is `G`, which means a rectangle drawn around the targets. The required argument following the `G` is the name of the gate, which is placed in the center of the rectangle. It is processed by Ti*k*Z, so it may include graphics commands (here, `\rotatebox`). Gate names may be enclosed in dollar signs to be typeset in math mode.

In addition to wires and gates, ⟨q|pic⟩ allows for several other common elements of circuit diagrams. The `=` command in line 6 takes one optional argument and draws it in the center of the circuit with a white background. If, as in this case, no argument is given, an equals sign is drawn.

The three remaining lines contain additional gates: two controlled NOTs and one Toffoli. One can use `C` for controlled NOT and `T` for Toffoli; for example, line 7 could have been written `b C c`, a controlled NOT with target b and control c. However, since these gates are so common, ⟨q|pic⟩ interprets a line containing only a list of wires as such a gate: `+` indicates that the operator $\oplus$ should be applied to that wire, and any other wires are treated as controls. Notice that the target may occur anywhere in the list. If no target is listed, all wires are drawn as controls, indicating a (controlled) Z gate.

2.2 Example 2: Quantum Fourier Transform

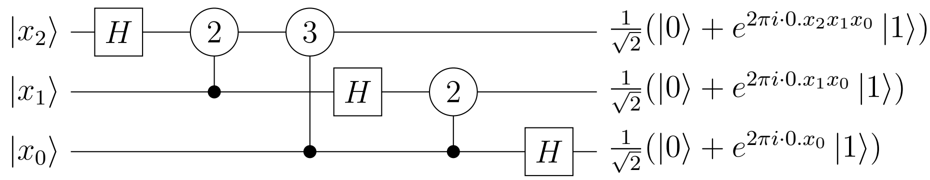
Our second example is a 3-bit QFT, shown in Figure 3.

The `PREAMBLE` commands insert LaTeX code before the Ti*k*Z environment; in this case, we define the commands `\ket` and `\phase`, which we will use in the wire labels. (`\providecommand` is useful for this construct, since it defines the command only if it is not already defined.)

The `SCALE` command scales the graphical elements in the picture, here by a factor of 1.5. Note that text is not scaled.

In lines 4–6 we declare the wires, as discussed in Section 2.1. Note that the internal wire names have nothing to do with the depiction in the diagram.

The remaining lines contain two types of gates: `H` (Hadamard) and `P` (phase shift). `H` is required to have one target, and `P` typically does as well. For a Hadamard, the `H` completely specifies the gate; for a phase shift (as with the rectangle in Section 2.1) we must also specify something to be written inside the circle. In this example the phase shifts have controls and the Hadamard gates do not, but either gate type is allowed to have an arbitrary number of controls.




```
PREAMBLE \providecommand{\ket}[1]{\left|#1\right\rangle}
PREAMBLE \providecommand{\phase}[1]{e^{2{\pi}i\cdot#1}}
SCALE 1.5
a2 W \ket{x_2} \frac{1}{\sqrt{2}}(\ket{0}+\phase{0.x_2x_1x_0}\ket{1})
a1 W \ket{x_1} \frac{1}{\sqrt{2}}(\ket{0}+\phase{0.x_1x_0}\ket{1})
a0 W \ket{x_0} \frac{1}{\sqrt{2}}(\ket{0}+\phase{0.x_0}\ket{1})
a2 H
a2 P $2$ a1
a2 P $3$ a0
a1 H
a1 P $2$ a0
a0 H
```

Figure 3: Quantum Fourier transform on three bits: diagram and code.

It is worth looking at the spacing between gates in Figure 3. Most of the gates are separated by horizontal space, but there is less space between the second phase gate and the Hadamard on $|x_1\rangle$. This is because ⟨q|pic⟩ detects that these two gates operate on disjoint sets of wires and may be performed in parallel. (If we changed the order of the wires, ⟨q|pic⟩ would draw the Hadamard directly above or below the phase shift; in Figure 3, the two gates are simply next to each other.)

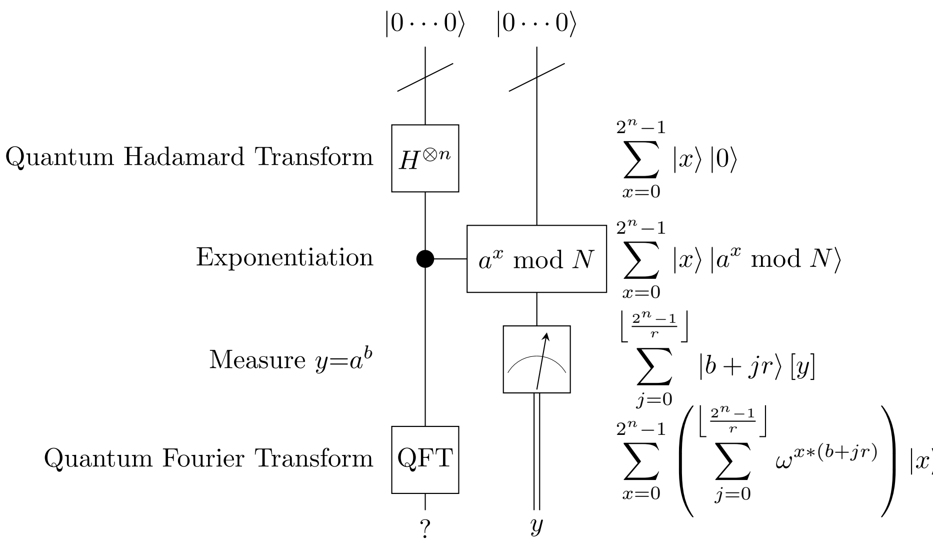
⟨q|pic⟩ uses a simple greedy algorithm to determine gate sequences: it divides the circuit into a series of time slices, or *slices*, placing each gate in the earliest possible slice. This is one of the key features of ⟨q|pic⟩; the user can specify the list of gates in logical order without worrying about exactly how they will appear on the page. Of course, there are ways to override this default behavior as needed; see Section 3.5 for more information.

2.3 Example 3: Shor's Algorithm

Figure 4 illustrates some additional features of ⟨q|pic⟩, and also a different application: a high-level schematic of an algorithm.

Line 1 contains the `VERTICAL` command; this tells ⟨q|pic⟩ to render the circuit vertically, with time flowing downward. By default the top and bottom labels on the wires are typeset horizontally; we could also indicate an angle (e.g., `VERTICAL 45`), where `0` is horizontal and `90` is vertical. The other new command at the top, `DEPTHPAD 3`, changes the spacing between slices to 3 points.

Lines 6 and 7 look like standard wire declarations, except for the `width=25` on line 7. This indicates that ⟨q|pic⟩ needs to leave more space for this wire on the page, to accomodate




```
1  VERTICAL
2  SCALE 2.1
3  DEPTHPAD 3
4  PREAMBLE \providecommand{\ket}[1]{\left| #1 \right\rangle}
5
6  x W \ket{0\cdots 0} ?
7  y W \ket{0\cdots 0} y width=25 # this wire needs to be wider
8
9  x y / width=10 height=5
10 x G $H^{{\otimes} n}$ %Quantum Hadamard Transform% $\displaystyle\sum_{x=0}^{2^n-1}\ket{x}\ket{0}$
11 y G $a^x \bmod N$ x width=25 % Exponentiation% $\displaystyle\sum_{x=0}^{2^n-1}\ket{x}\ket{a^x\bmod N}$
12 y M % Measure $y{=}a^b$% $\displaystyle\sum_{j=0}^{\left\lfloor\frac{2^n-1}{r}\right\rfloor}\ket{b+jr}[y]$
13 x TOUCH
14 x G QFT %Quantum Fourier Transform% $\displaystyle\sum_{x=0}^{2^n-1}\left(\sum_{j=0}^{\left\lfloor\frac{2^n-1}{r}\right\rfloor}\omega^{x*(b+jr)}\right)\ket{x}$
```

Figure 4: Shor's algorithm: diagram and code.

the wider exponentiation gate. As we will discuss in Section 3.3, we could also have said `breadth=25`. Note that `#` is a comment character in ⟨q|pic⟩; the remainder of the line is discarded.

Line 9 draws the slashes on the wires (a common way to indicate that a wire carries more than one qubit); the list of wires comes first, and the optional argument at the end is written in math mode in the diagram. (For ⟨q|pic⟩'s purposes this is a "gate", even though it does not correspond to any computation.) This line also contains specifications for the width and height of the gate; these can be applied to any gate to override the defaults, most commonly to accommodate extra text.

The next few lines incorporate another ⟨q|pic⟩ capability: comments. Text after a percent sign is written to the left of the circuit (or above a horizontal circuit); if there is a second percent sign, that text is written to the right of (or below) the circuit. As we see in this example, the comments are processed by LaTeX. Long comments tend to work better with vertical circuits.

Aside from the comments, lines 10, 11, and 14 contain no new ideas; we have several `G` gates, one with a control and a width parameter. Line 12 introduces a measurement operator; this draws a meter on each specified wire. If an optional argument follows the `M`, the meter is replaced by a bullet shape with the specified text inside; see Section 3.4 for an example.

A ⟨q|pic⟩ wire can have one of three types: `qwire` (quantum, or single line), `cwire` (classical, or double line), and `owire` (off). Since measurement changes a wire from quantum to classical, the `M` operator automatically changes any affected wires from `qwire` to `cwire`. As we will see in Section 2.4, we can also change wire types (and colors and styles) directly.

As noted in Section 2.2, ⟨q|pic⟩ tries to place gates within the same slice when possible—for example, the gates on lines 12 and 14 could be drawn next to each other. In this diagram the comments would then be on top of each other, which would look bad. The solution is line 13, `x TOUCH`, which is essentially a "no-op". It tells ⟨q|pic⟩ to "touch" the `x` wire at this time (i.e., during the measurement), forcing the final gate to occur in a later slice. `TOUCH` with no arguments would touch all wires. See Section 3.5 for some other ways to manipulate which gates occur at the same time.

2.4 Example 4: Teleportation

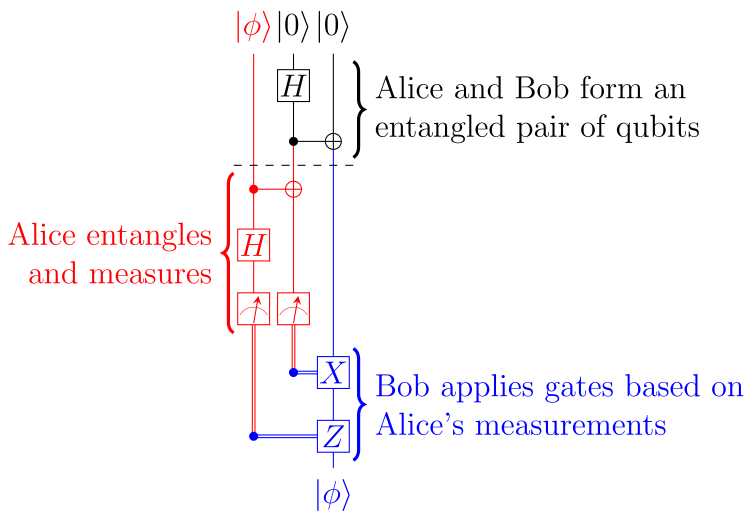
In Figure 5 we introduce a few more things ⟨q|pic⟩ can do: color, changing wire styles with a colon, and specifying ranges of slices.

The first use of the `color=` syntax is when we declare wire `0` in line 5. This sets the (initial) color of the wire to red. Similarly, line 13 specifies a red gate. One can use the similar `style=` syntax to pass style commands to Ti*k*Z; this can change the thickness of a wire or gate. (See Section 3.3 for details.)

For Bob's gates on lines 17 and 18, we instead use a macro, `Bob`, which we defined in line 3. Roughly speaking, the `DEFINE` command tells ⟨q|pic⟩ to replace the next word (`Bob`) with the rest of the line (`color=blue`) wherever it is used. (Note that this expansion affects ⟨q|pic⟩ commands, but not the comments on lines 11 and 23.) In the bottom two gates, the connectors are blue (the gate color) but drawn with double lines (since the controls are classical).

Line 10 uses the `:` syntax to change the colors of the two wires. This means that, as of this gate, `color=red` is applied to wire 1, and `Bob` (i.e., `color=blue`) is applied to wire 2. Lines 17 and 18 use the same syntax to change the wire type to `owire`, which means those wire are "off" (not drawn) starting from those points.

The remaining commands refer to slices; the first set of gates is considered slice 0, then slice 1, and so on. In line 11, `@ 2` specifies the most recent 2 slices (i.e., 0 and 1). This command does nothing by itself, but it lets us attach a comment to this part of the circuit. In lines 16 and 19 the two numbers following the `@` are the first and last slices in the range. Be aware that ⟨q|pic⟩ is happy to draw comments on top of each other; it is up to the user to position them appropriately.




```
1  VERTICAL
2  PREAMBLE \providecommand{\ket}[1]{\left|#1\right\rangle}
3  DEFINE Bob color=blue
4
5  0 W \ket{\phi} color=red
6  1 W \ket{0}
7  2 W \ket{0} \ket{\phi}
8
9  1 H
10 +2:Bob 1:color=red
11 @ 2 %% Alice and Bob form an \\ entangled pair of qubits
12 CUT 2
13 +1 0 color=red
14 0 H color=red
15 0 1 M color=red
16 @ 2 4 color=red % Alice entangles and measures
17 2 X 1:owire Bob
18 2 Z 0:owire Bob
19 @ 5 6 Bob %% Bob applies gates based on Alice's measurements
```

Figure 5: Teleportation from Alice to Bob: diagram and code.

The CUT command tells ⟨q|pic⟩ to put "cut here" lines before the specified slices; in line 12, we place a single line before slice 2. If no arguments are given, CUT draws dashed lines between all pairs of slices.

3 ⟨q|pic⟩ Commands

Where Section 2 serves as a tutorial, this section serves as a reference. We give a complete description of all commands in ⟨q|pic⟩, sorted by categories. Where applicable, we give an example together with a small diagram; in all cases, the example is a valid input that generates the accompanying diagram.

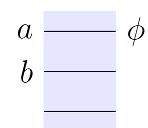
Throughout, we use “gate” in the ⟨q|pic⟩ sense to mean something that is drawn on a wire, whether or not it corresponds to some quantum operation.

3.1 Wires

3.1.1 Wire Declarations

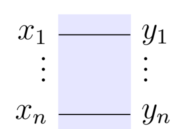
`name W [labels]` Declare a wire with the given name. If one label is given, it is used at the start; if two labels are given, the second is used at the end. Additional labels are normally ignored, but may appear if `START` and `END` are used (see Section 3.4). Subsequent declarations of the same wire simply append additional labels (if any) to the wire's list.

```
a W a \phi
b W b
c W
```



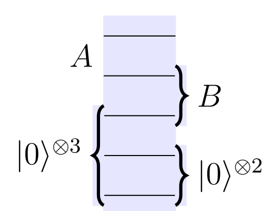
`...name W` Declare an ellipsis wire with the given name.

```
a W x_1 y_1
...b W
c W x_n y_n
```



`names W [labels]` The specified labels are applied to all the wires listed. If a label begins and/or ends with `<` or `>`, it is instead drawn with an open or close brace before and/or after it.

```
a0 a1 W A
b0 b1 b2 W |0\rangle^{\otimes 3}<
a1 b0 W >B
b1 b2 W >|0\rangle^{\otimes 2}
```



The wire name does not appear in the diagram, but is used by ⟨q|pic⟩ to identify the wire. One useful convention is for wire names to be made up of numbers, lowercase letters, and underscores. Technically, a wire name can be any non-whitespace string, except:

- Wire names should not contain any of the characters that ⟨q|pic⟩ handles specially: `\#${}%=@":;.`
- Any name of a built-in or user-defined ⟨q|pic⟩ command is a reserved word and may not be used.

- A wire name may not begin with `-` or `+`. A wire name preceded by a minus sign is interpreted as a negated control; one preceded by a plus sign is interpreted as a target.

- If a wire name starts with `...` it is treated as an ellipsis.

Wires appear in the diagram in the order they are declared. If the same wire is declared more than once, later declarations are ignored, except that additional labels (if any) are appended to the list of labels associated to the wire, as in the example above.

3.1.2 Undeclared Wires

⟨q|pic⟩ also allows the option of undeclared wires. This is useful for larger, computer-generated circuits, where you don't need the extra check to catch typos. By default, ⟨q|pic⟩ starts in "autowires" mode, allowing these undeclared wires. As soon as a single `W` declaration is seen, ⟨q|pic⟩ switches to a "declared" mode where all wires must be explicitly declared. A typical file will use one of the two modes exclusively.

- A valid name for an undeclared wire is either non-negative integer, or a string of the form

  ```
  lowers[_][num[,num[,num...]]]
  ```

  That is, a lowercase string, an optional underscore, and then a (possibly empty) comma-separated list of numbers.

- Undeclared wires appear on the page after any declared wires. Wires with integer labels are listed in numerical order; other undeclared wires are then sorted first by lowercase string, then by number of subscripts, and then lexicographically.

- An undeclared wire with an integer name has no label. Otherwise, an undeclared wire is given a starting label: the lowercase string, followed by the numbers as subscripts.

- Note that ⟨q|pic⟩ interprets the undeclared wire name in terms of a standard representation. If declared, `a_0` and `a00` are different strings and refer to different wires, but if undeclared they have the same representation and refer to the same wire.

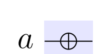
- The command `AUTOWIRES` sets ⟨q|pic⟩ to "autowires" mode for the rest of the circuit. This lets you mix declared and undeclared wires.

3.2 Gates

3.2.1 Controlled NOT and controlled Z

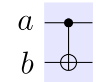
`target N` Negate the target wire.

```
a N
```



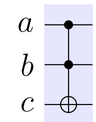
`target C control` Controlled NOT.

```
b C a
```



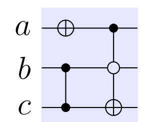
`target T control1 control2` Toffoli gate.

```
c T a b
```



`controls` If a list of wires is given without a gate, it is assumed to be a (generalized) controlled Z or NOT. A target of NOT is specified with a `+`.

```
+a
b c
a -b +c
```



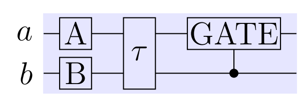
Note the negated control in the last example. In general, any time a wire is used as a control, we can use a `-` to change it to a negated control.

3.2.2 General Gates

⟨q|pic⟩ provides two ways to do a general gate: `G` for rectangles and `P` for circles (or, more precisely, ellipses). In theory, each of these can be used with the right attributes (Section 3.3) to simulate the other, but their default behavior is different.

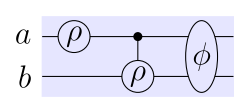
`targets G name [controls]` General rectangular gate. It must have at least one target.

```
a G A
b G B
a b G $\tau$
a G GATE b width=35
```



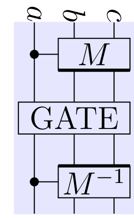
`targets P name [controls]` Circular gate. It typically has one target but may have more.

```
a P $\rho$
b P $\rho$ a
a b P $\phi$
```



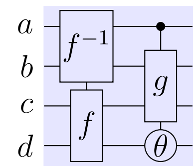
`targets G| name [controls]` **or** `targets |G name [controls]` The same idea as `G`, but the indicated side of the rectangle is thicker to indicate directionality.

```
VERTICAL 90
b c G| $M$ a
a b c G GATE
b c |G $M^{-1}$ a
```



`targets G name targets G name ...` One can combine more than one `G`, `P`, `G|`, or `|G` into a single gate. Each subsequent rectangle (or ellipse) is applied to all wires since the previous one.

```
a b G:width=20 $f^{-1}$ c d G $f$
b c G $g$ d P $\theta$ a
```

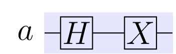


In this last example, note that `:width=20` applies only to that rectangle, but the other has the default width.

3.2.3 Other predefined Gates

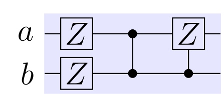
`target H [controls]` **or** `target X [controls]` Hadamard or X gate. It must have exactly one target.

```
a H
a X
```



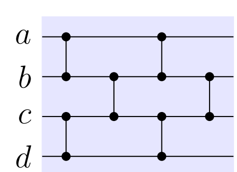
`target Z [controls]` Z gate. As with Hadamard and X, it must have exactly one target. However, controlled Z can also be depicted with dots.

```
a Z
b Z
a b
a Z b
```



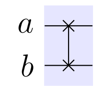
It is worth noting that a two-wire gate with no controls is also useful for sorting networks. (See Section 3.6 for the `R` command.)

```
a b
c d
b c
R 0 1
```



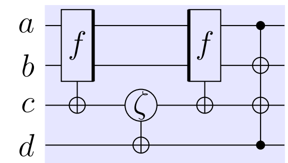
`target1 target2 SWAP [controls]` Swap gate.

```
a b SWAP
```



`+` Any time a wire would normally be a control but is preceded by `+`, it is drawn as a target.

```
a b G| $f$ +c
c P $\zeta$ +d
a b |G $f$ +c
a +b +c d
```



3.3 Attributes

Attributes are of the form `attribute=value`, and can affect different properties of a wire or gate:

- Size: `height`, `width`, `length`, `breadth`, `size`.
- Appearance: `color`, `fill`, `style`.
- Other properties: `type`, `hyperlink`, `operator`, `shape`.

Attribute names may be abbreviated; e.g., `oper` for `operator` or `co` for `color`. At least two letters must be used.

Attributes can be used in one of three ways:

- Anywhere on a line, separated from other elements by spaces. This represents a property of the gate (or wire, if it's a `W` declaration).
- After to a gate or other circuit element, separated by a colon. This represents a property of that particular element.
- After a wire name in a gate, separated by a colon. This represents a change in the wire's properties, effective as of that gate. If this part of the circuit is repeated or reversed (see Section 3.6), ⟨q|pic⟩ will try to repeat or undo the change.

Note that not all attributes can be applied in all situations.

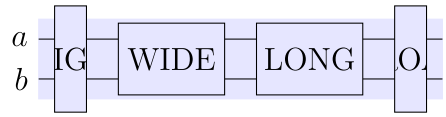
3.3.1 Size Attributes

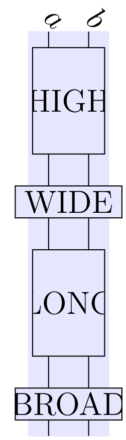
There are two different coordinate systems for specifying gate sizes: *height* v. *width* and *length* v. *breadth*. *Height* is always vertical on the page, and *width* is horizontal; for example, if a rectangle needs to be larger to fit multiple characters, one can increase its width. *Length* refers to the direction of time, and *breadth* refers to the perpendicular direction; for example, if a rectangle needs to be larger to convey that it takes a long time, one can increase its length.[3]

[3] For horizontal circuits, length is width and breadth is height. For vertical circuits, length is height and breadth is width.

`height=value`, `width=value`, `length=value`, `breadth=value` Change the size of a gate.[4] Units are in points (although scaling may change this); the default value for a rectangle is 12 in both directions. ⟨q|pic⟩ will treat the specified value as a minimum and increase it if necessary to span the indicated wires.

```
a b G HIGH height=40
a b G WIDE width=40
a b G LONG length=40
a b G BROAD breadth=40
```

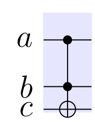





```
VERTICAL 45
a b G HIGH height=40
a b G WIDE width=40
a b G LONG length=40
a b G BROAD breadth=40
```

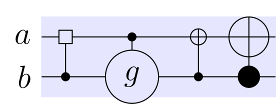
`W breadth=value` Set the breadth of a wire. The breadth can only be changed in the initial declaration, not during the circuit. You can instead use `width` in a vertical circuit or `height` in a horizontal circuit. You cannot set the length of a wire; it is determined by the circuit. Wires are automatically separated by a distance of `WIREPAD`, which defaults to 3 (Section 3.10).

```
a W a breadth=20
b W b
c W c breadth=2
a b +c
```



`size=value` Change the height and width of a gate simultaneously. If applied to a target or control wire this instead changes the size of the target or control.

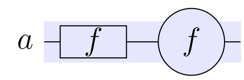
```
a G {} size=5 b
b P $g$ size=20 a
+a b
+a:size=15 b:size=8
```



[4]For controlled NOT or Z, length affects only spacing.

target P width=value [controls] The only difference between `P` and `G` is how they are affected by size changes. ⟨q|pic⟩ tries to keep a single-target `P` a circle; changing height or width affects both. To make it an ellipse, one needs to give both attributes explicitly.

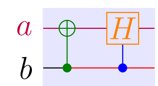
```
a G $f$ width=25
a P $f$ width=25
```



3.3.2 Appearance Attributes

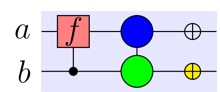
color=value Change the color of a gate or wire.

```
color=purple a W a
b W b
+a color=green!50!black b:color=red
a H:color=orange b color=blue
```



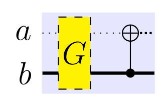
fill=value Change the fill color of part of a gate. Wires and controlled nots do not have fills, but targets can.

```
a G $f$ b fill=red!50!white
a P:fill=blue {} b P:fill=green {}
+a fill=yellow # applies to (absent) gate
+b:fill=yellow # applies to target
```



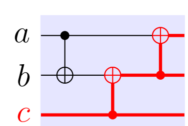
style=value Change the style of a gate or wire. The style parameter is passed directly to Ti*k*Z, except that underscores are replaced by spaces. Multiple styles can be separated by commas (and are passed to Ti*k*Z as a group); multiple attributes are separated by colons.

```
style=dotted a W a
b W b style=very_thick
a b G style=dashed:fill=yellow $G$
+a b a:style=densely_dotted,thick
```



The last gate above illustrates another rule: If a wire is specified multiple times in a single gate, only the first appearance matters to draw the gate, but the later ones can be used to attach attributes. This can make the code cleaner.

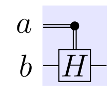
```
DEFINE on color=red:style=very_thick
a W a
b W b
c W c on
+b a
+b c on b:on
+a b on a:on
```



3.3.3 Other attributes

`type=value`, `qwire`, `cwire`, `owire` There are three types of wires: quantum (single line), classical (double line), or off (no line). By default, all wires are quantum. You can set or change the type using the `type` command (⟨q|pic⟩ looks only at the first letter of value) or using the shorthand `qwire`, `cwire`, `owire`. (These have the same syntax with colons as `attribute=value`.) `type` applies only to wires, but the quantum/classical status of wires affects how gates are drawn.

```
a W a cwire
b W b
b H a:owire
```

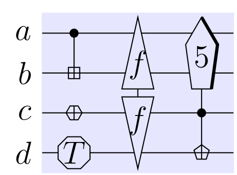


`shape=value` You can change the shape of a G, P, or target. Value is a integer, or possibly `box`, `circle`, `<`, or `>`:

0	no boundary
1	control
-1	negated control
`circle`, 2	circle
`box`, n	n-sided polygon with a flat bottom/right (`box` $= 4$) (for $n \geq 3$)
$-n$	n-sided polygon with a vertex on the bottom/right (for $n \geq 3$)
`<`, `>`	triangle pointing forward or backward

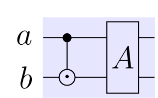
⟨q|pic⟩ will draw a regular polygon on a single wire, or stretch the polygon if it spans multiple wires or if a size is specified. In horizontal mode, the shape is drawn either with a side or a vertex on the bottom; in vertical mode, the side or vertex is on the right of the circuit. If not attached to any particular element, `shape` affects any G or P on the line, or (if there is no G or P) the target.

```
+b a shape=box
+c shape=6
d P $T$ shape=8
a b G:shape=3 $f$ c d G:shape=-3 $f$
a b G| 5 c +d shape=5
```



`operator=value` This can be used to specify the operator for `G`, `P`, `M`, or a target. For `G`, `P`, `M`, this is an alternative to having the operator be the next item on the line.

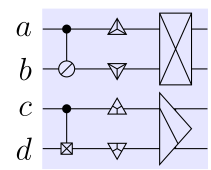
```
+b a operator=$\cdot$
b a G:operator=$A$
```



Certain values of `operator` are treated specially. (Note the need for a double backslash.)

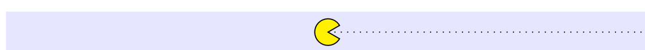
`0`	none
`- | / \\`	a single line
`+ x X`	two perpendicular lines
`*`	four lines if `shape` is 2, 4, or −4, otherwise one line to each vertex
`-*`	same as `*`, but one line to each side.

```
+b a op=/
+d c shape=4 op=x
+a shape=3 op=* size=8
+b shape=-3 op=* size=8
+c shape=3 op=-* size=8
+d shape=-3 op=-* size=8
a b G:op=x
c d G:op=\\:shape=>
```



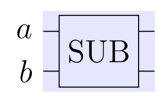
Finally, if the operator is enclosed in double quotes, it is drawn by Ti*k*Z (with (0,0) as the center of the desired location):

```
a W owire
a LABEL width=100
a:op="\draw[fill=yellow] (0,0) -- (30:5pt) arc (30:330:5pt) -- cycle;":sh=0:style=dotted:qwire
a LABEL width=100
```



`hyperlink=value` This specifies that a gate[5] is a hyperlink to a target elsewhere in the document, possibly declared with `HYPERTARGET` (see Section 3.9).

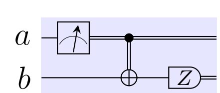
```
b a G SUB width=30 hyperlink=hyper_qpic_target
```



3.4 Measurement and Other Wire Type Changes

`wires M [name]` Measure the wires. If the optional argument is given, ⟨q|pic⟩ draws a D-shaped "bullet" containing the name; if not, ⟨q|pic⟩ draws a meter. Measurement automatically changes its targets to `cwire`.

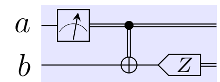
```
a M
+b a
b M {\scriptsize $Z$}
```



[5]There is a known bug: in some older versions of TEX, only one edge of the gate is the link.

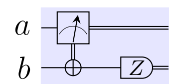
`MEASURESHAPE tag` Change the default shape of the measurement gate. `MEASURESHAPE` must be followed by `D` (the default) or `tag`. Meters are unaffected. Only one shape may be used throughout a circuit.

```
MEASURESHAPE tag
a M
+b a
b M {\scriptsize $Z$}
```



`wire:cwire` If a wire is changed to a classical wire, ⟨q|pic⟩ draws a meter in place of a control.

```
a:cwire +b
b M {\scriptsize $Z$}
```

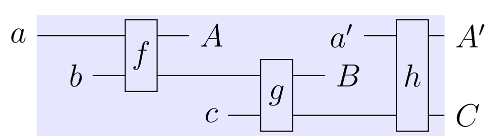


`target OUT value` **or** `target IN value` Drop a wire out or bring it back in, generally because it has a known value. The wire type is changed to `owire` (by `OUT`) or `qwire` (by `IN`).

```
a OUT 0
LABEL
a IN 1
```

`target(s) START` **or** `target(s) END` Start or end a wire later in the circuit, possibly because it is not relevant at certain times. The labels are not specified by these commands, but in a `W` declaration. The wire type is changed to `owire` (by `END`) or `qwire` (by `START`). If the first of these gates to apply to a wire is `START`, then its initial start is deferred.

```
a W a A a' A'
b W b B
c W c C
b START
b a G $f$
a END
c START
c b G $g$
b END
a START
a c G $h$
```



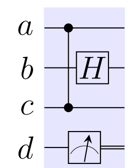
Certain commands (e.g., `TOUCH`, `LABEL`) apply by default to all wires. More precisely, wires that are not currently "active" (as determined by `START` and `END`) are excluded.

3.5 Managing Slices

⟨q|pic⟩ greedily divides a circuit into *slices*. Each gate is placed into the earliest possible slice[6], with the constraint that different gates applying to the same wire must occur (in order) in different slices. Within a slice, ⟨q|pic⟩ arranges gates as efficiently as possible without overlapping on the page. ⟨q|pic⟩ places padding to separate slices, but subslices are immediately adjacent to one another.

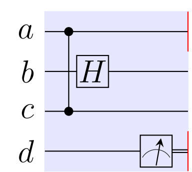
Most of the time, ⟨q|pic⟩'s default behavior will look fine. The commands in this section can be used to modify this behavior.

```
a c
d M
b H
```



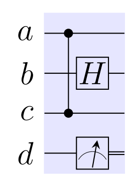
`[wires] TOUCH` Pretend that the given wires were "touched" by the most recent gate (or the last time any of its targets was used, whichever is later), forcing subsequent gates to be in later slices. If no wires are given, touch all wires. Technically, `TOUCH` draws an invisible line, which can be made visible with attributes.

```
a c
d TOUCH
d M
b H
a d TOUCH color=red
```



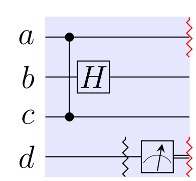
`[wires] PHANTOM` The same idea as `TOUCH`, but with subslices rather than slices.

```
a c
d PHANTOM
d M
b H
```



`[wires] BARRIER` The same idea as `TOUCH` (i.e., affecting slices), but with a zigzag line.

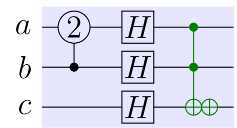
```
a c
d BARRIER
d M
b H
a d BARRIER color=red
```



`LB` **and** `LE` Begin and end a *level*: a set of gates required to be in the same slice. No checking is done to see whether gates in a level use the same wires. Attributes placed on the `LB` line will be passed on to the gates.

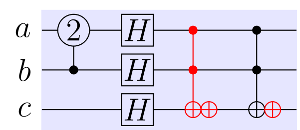
[6] Except that `START` is always as late as possible.

```
a P 2 b
LB
a H
b H
c H
LE
LB color=green!50!black
c T a b
c N
LE
```



; Multiple gates may be listed on one line, separated by semicolons. They will all be at the same depth; more precisely, ⟨q|pic⟩ will enclose them in `LB` and `LE`. If any command (typically the first or last) contains nothing but an attribute, that attribute is passed on to all gates.

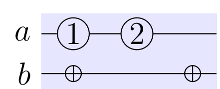
```
a P 2 b
a H ; b H ; c H
c T a b ; c N; color=red
c T a b ; c N color=red
```



`MARK name(s)` Place a "mark" with the given name at the depth of the most recent gate in the circuit. This has no effect by itself, but the marks can be used as arguments to `R` (Section 3.6) or `@` (Section 3.8).

`MIXGATES value` If `MIXGATES` is set to 0, ⟨q|pic⟩ will not mix different types of gates within the same slice. The default value is 1.

```
a P 1
+b
MIXGATES 0
a P 2
+b
```

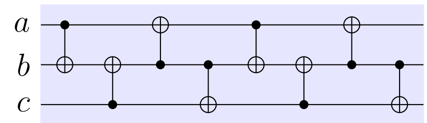


3.6 Reversing and Repeating

One common quantum operation is to reverse or repeat part of a circuit. In ⟨q|pic⟩, this can be done by selecting sets of slices. The first slice in the circuit is 0, then 1, and so on. For the `R` command, -1 refers to the most recent slice, -2 to the one before that, and so on.

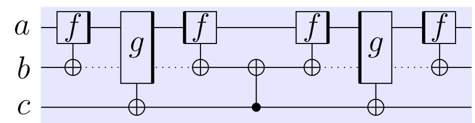
`R start end` If `start` $<$ `end`, repeat the specified range of slices (those between `start` and `end`, including both endpoints). We replay the slices in order.

```
+b a
+b c
+a b
+c b
R 0 3
```



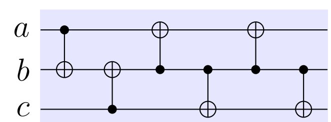
R [start end] If start $\geq$ end, reverse the specified range of slices (those between end and start, including both endpoints). We replay the slices in reverse order, swapping G| with |G and undoing wire changes. If the endpoints are not included, we take start to be -2 and end to be 0 (i.e., we undo everything but the most recent slice).

```
a G| $f$ +b:style=dotted
a b G| $g$ +c
R 0 0
c +b
R
```



R [name name] We can also use names set by the MARK command in place of numbers. We will replay the slices either forward or in reverse, depending on whether the first or second mark is the earlier one. If names are used, the earlier mark is *excluded*. (You can also specify one slice with a mark and one with a number; again, if the earlier slice is a mark, it is excluded.)

```
+b a
+b c
MARK m0
+a b
+c b
MARK m1
R m0 m1
```

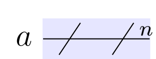


3.7 Other Circuit Elements

These next few items are all considered "gates" by ⟨q|pic⟩. They do not correspond to any computation, but they take up space on the page and are assigned to slices.

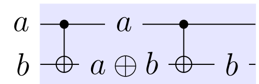
wires / [name] Draw a slash across each wire; the optional argument is written (in math mode) next to the slash. This is sometimes used to denote the number of qubits represented by a wire.

```
a /
a / n
```



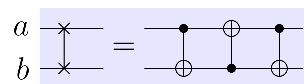
[targets] LABEL [labels] Place the given labels (in math mode) on the wires. If no targets are given, the command applies to all wires. There should be (1) one label for each wire, or (2) just one label (to be repeated on each wires), or (3) no labels (the empty label is placed on all wires). The label `...` is turned into `\cdots`. In vertical circuits, labels are rotated by 90 degrees. The empty label is sometimes useful to pad with space, especially in conjunction with a `length` attribute.

```
+b a
LABEL a {a \oplus b}
+b a
b LABEL b width=6
```



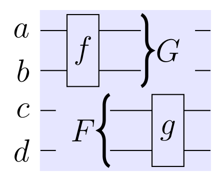
[targets] = [label] Place a single label, centered across the specified targets. If no targets are specified, use all wires. If no label is specified, use =.

```
a b SWAP
=
a +b
+a b
a +b
```



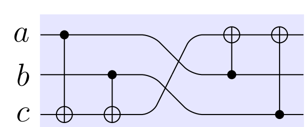
[targets] [<|>]=[<|>] [label] If `=` is preceded and/or followed by `<` or `>`, then a curly brace is drawn before and/or after the label, in the indicated direction.

```
a b G $f$
a b >= $G$ width=20
c d =< $F$ width=20
c d G $g$
```



wires PERMUTE Change the order of the specified wires on the page. The left-to-right order on the line will be the new top-to-bottom (or left-to-right) order. ⟨q|pic⟩ will draw smooth lines for each wire, with the rounded corners specified by the global parameter `CORNERS` (Section 3.10).

```
+c a
+c b
c a b PERMUTE
R 0 1
```

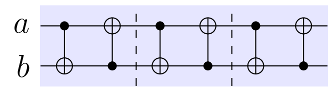


3.8 Comments

These next few commands are not considered "gates" by ⟨q|pic⟩. They do not take up any space in their circuit, and their presence does not affect the locations of other elements.

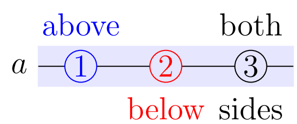
`CUT [slices]` Place “cut here” (dashed) lines just before each specified slice. If no slices are given, place cuts in all possible places.

```
a +b
+a b
R 0 1
R 0 1
CUT 2 4
```



`% [comment1] [% comment2]` Place comments next to the gate (there must be a gate on the same line). The `comment1` is placed above or to the left of the circuit; `comment2` is placed below or to the right.

```
a P 1 color=blue % above
a P 2 color=red %% below
a P 3 % both % sides
# separate the slices
DEPTHPAD 10
```

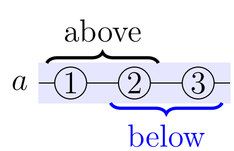


Note that there are two types of comments in ⟨q|pic⟩: comments in the diagram, indicated by `%` and passed on to LaTeX, and comments within ⟨q|pic⟩ code itself, which start with `#` and are discarded by the parser.

`[wires] @ [num1 [num2]]` Specify a rectangular region. The “breadth” dimension spans the specified wires; if none are listed, it spans all wires. The “length” dimension is determined by the arguments after the `@`. With no arguments, it spans the most recent slice; with one, it refers to the most recent `num1` slices; with two, it refers to slices `num1` through `num2` (including both endpoints). (As with `R`, a name set by `MARK` may be used for either or both endpoints, and if the earlier slice is specified by a name it is excluded.)

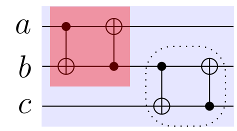
This region can serve as a placeholder for comments, in which case the wires are usually not specified.

```
a P 1
a P 2
a P 3
@ 0 1 % above
@ 2 color=blue %% below
```



The region can also be made visible using attributes like `fill`, `style`, and `color`.

```
+b a
+a b
+c b
+b c
a b @ 0 1 fill=red
b c @ 2 3 color=black style=dotted,rounded_corners=10pt
```

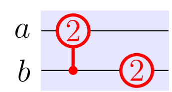


3.9 Macros and LaTeX Code

The next few commands are for defining macros or for placing LaTeX code before and after the Ti*k*Z code generated by ⟨q|pic⟩. Each may be repeated as often as necessary.

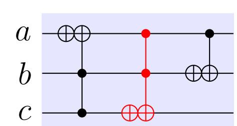
`DEFINE macro text` Creates user-defined ⟨q|pic⟩ code. Hereafter, wherever the word `macro` appears, it will be replaced by `text`.

```
DEFINE loud color=red:style=very_thick
DEFINE phase P 2 loud
a phase b
b phase
```



`args DEFINE macro text` If `DEFINE` has n arguments, then the n space-delimited words preceding `macro` will be used to replace those arguments wherever they occur (as space- or colon- or semicolon- delimited words) within `text`.

```
x y z DEFINE Noffoli x N ; x T y z ;
a b c Noffoli
c a b Noffoli color=red
b a a Noffoli # repeated a ignored
```



`COLOR name r g b` Insert a `\definecolor` command defining `name` with the specified rgb values (which should be between 0 and 1).

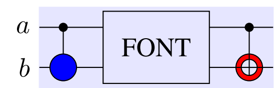
`HEADER text` Add a line which will be interpreted by `tikz2preview` to place `text` at the start of its output, before the `\begin{document}`.

`PREAMBLE text` Add `text` to the LaTeX code just before the tikzpicture.

`PRETIKZ text` Add `text` at the start of the tikzpicture, before the diagram code.

`POSTTIKZ text` Add `text` at the end of the tikzpicture, after the diagram code.

```
HEADER \usepackage{times}
PREAMBLE \newcommand{\name}{FONT}
PRETIKZ {\draw[fill=red] (79,0) circle(5pt);}
POSTTIKZ {\draw[fill=blue] (9,0) circle(5pt);}
a +b
a b G \name width=40
a +b
```



`HYPERTARGET name` Add `\hypertarget{name}{}` at the start of the tikzpicture, to create a hyperref target (say, for use by `hyperlink`).

```
HYPERTARGET hyper_qpic_target
b a G DETAILS width=50
```

One of the design principles behind ⟨q|pic⟩ is that it should solve common problems, but not all problems. If a user would like to include something that can't be drawn using the standard commands, the full power of Ti*k*Z can be accessed via `PRETIKZ` and `POSTTIKZ` (and via `operator=''...''` in Section 3.3.3. For this reason, ⟨q|pic⟩ produces commented Ti*k*Z code, to make it easy to find what coordinates should be used to draw extra elements.

3.10 Global Parameters

The remaining commands set global parameters. They can be used multiple times, but only the final instance will have any effect.[7]

Some commands use ⟨q|pic⟩'s internal unit, which is 1 point (or 1/12 inch). Applying `SCALE` changes this unit size.

`VERTICAL [deg1 [deg2]]` Change the flow of time to vertical. Starting labels are rotated by `deg1` degrees, ending labels by `deg2` degrees; values should be between 0 and 90. If `deg2` is absent, `deg1` is used; if both are missing, the default is 0° (horizontal).

`HORIZONTAL` Change the flow of time to horizontal. This is the default.

`DEPTHPAD value` Set the amount of space between slices. The default is 6.

`WIREPAD value` Set the amount of space between wires. The default is 3.

`GATESIZE value` Set the default height and width of a gate. The default is 12.

`COMMENTSIZE value` Set the width of the comment region outside the circuit. The default is 144.

[7]One exception: If `VERTICAL` and `HORIZONTAL` are used throughout the circuit, this may confuse ⟨q|pic⟩'s interpretations of `height=` and `width=` attributes.

`SCALE value` Set an overall scale factor for graphical elements in the circuit (but not for text). The default is 1.

`MEASURESHAPE value` Set the shape of an measurement with a label. `value` can be `D` or `tag`. The default is `D`.

`CORNERS value` Set the argument to Ti*k*Z's `rounded corners` when wires are permuted (0 means no rounding). The default is 4.

`OPACITY value` Set the fill opacity for rectangles drawn with `@`. The default is 0.2.

`WIRES value` Prepend `value` to all wire labels within math mode (e.g., `\scriptstyle`). The default is to prepend nothing.

`PREMATH value` Prepend `value` to all wire labels before math mode. The default is to prepend nothing.

`POSTMATH value` Append `value` to all wire labels after math mode. The default is to append nothing.

`BGCOLOR [value]` Set the background color for the diagram. The default is white; however, if `BGCOLOR` is called with no argument, it sets the background to `bg`. This color is not defined in standard LaTeX, but in BEAMER it is always equal to the background of the current template.

4 Running `qpic`

4.1 Installation

`qpic` is available on both `github` and `pypi`. Installation is recommended using `pip` (included in Python since 2.7.8 and 3.4). `pip` can be installed for earlier versions of Python (2.6 and 3.3). Using `pip`, `qpic` can be installed by

```
pip install qpic
```

If you wish to install `qpic` as a single user, add `$HOME/.local/bin` to your `$PATH` and install using

```
pip install --user qpic
```

4.2 Basic usage

`qpic` generates Ti*k*Z code by default. `qpic` can also generate `pdf` files using `pdflatex` and `png` files using `convert` (from `ImageMagic`). The following commands both send Ti*k*Z code generated from `diagram.qpic` to `<stdout>`.

```
qpic diagram.qpic
qpic < diagram.qpic
```

4.3 Choosing file type with `-f`

The `-f, --filetype` flag indicates the desired output file type. Valid options are `tikz`, `tex`, `pdf` or `png`. Instead of using `<stdout>`, qpic creates a file. The output filename is deduced from the input filename. If the `qpic` code is provided via `<stdin>` the default output basename is `texput`.

```
qpic diagram.qpic -f tikz # Creates diagram.tikz
qpic diagram.qpic -f tex # Creates diagram.tex
qpic diagram.qpic -f pdf # Creates diagram.pdf
qpic diagram.qpic -f png # Creates diagram.png
```

4.4 Choosing output file with `-o`

The `-o, --output` flag indicates the desired output filename. If the filename ends with a valid suffix, `qpic` creates a file of that type.

```
qpic diagram.qpic -o other.tikz # Creates other.tikz
qpic diagram.qpic -o other.tex # Creates other.tex
qpic diagram.qpic -o other.pdf # Creates other.pdf
qpic diagram.qpic -o other.png # Creates other.png
```

Mixing `-f` and `-o` options is allowed, and `qpic` tries to do what the authors think is reasonable, which is not guaranteed to correspond with what you think is reasonable.

5 `qpic` and LaTeX

There are three major use-case scenarios for the `qpic` program:

1. Create standalone PDF or PNG graphics from ⟨q|pic⟩ files.
2. Include ⟨q|pic⟩ diagrams as PDF graphics in a LaTeX file.
3. Include ⟨q|pic⟩ diagrams as Ti*k*Z code in a LaTeX file.

5.1 Include ⟨q|pic⟩ Diagrams as PDF Graphics in a LaTeX File

The package `graphicx` is required to include the PDF graphics in LaTeX documents. Add the following line to `yourfile.tex` before `\begin{document}`.

```
\usepackage{graphicx}
```

The ⟨q|pic⟩ PDF graphic `diagram.pdf` is included in the document using the command:

```
\includegraphics{diagram}
```

5.2 Include ⟨q|pic⟩ Diagrams as Ti*k*Z Code in a LaTeX File

The package `tikz` is required to compile Ti*k*Z code in LaTeX documents. Add the following line to `yourfile.tex` before `\begin{document}`.

```
\usepackage{tikz}
```

The ⟨q|pic⟩ Ti*k*Z code `diagram.tikz` is included in the document using the command:

```
\include{diagram.tikz}
```

5.3 Comparing PDF and Ti*k*Z Inclusion Methods

These two approaches to including ⟨q|pic⟩ diagrams in LaTeX files each have their strengths and weaknesses. Both were used in the preparation of this document.

Advantages of PDF inclusion:

- Only modified graphics need to be recompiled, resulting in a faster LaTeX compilation. With Ti*k*Z inclusion, every Ti*k*Z graphic must be reconstructed as part of the document build process.
- Graphics can be scaled using the `\includegraphics` command. This scaling is independent of the ⟨q|pic⟩ scaling and makes it easier to generate graphics of a specific size.

Advantages of Ti*k*Z inclusion:

- The graphics are aware of the document settings, including font style, when they are created. Thus a slightly different graphic is created from the same ⟨q|pic⟩ file if it is in a slide environment as opposed to a document environment.
- On systems where pdfLaTeX is not available (or is not recent enough to support the `preview` environment), PDF inclusion is not possible and Ti*k*Z inclusion must be used.

A Tokenizing

The parsing rules for ⟨q|pic⟩ can be a bit confusing. Both `#` and `%` are used to delimit different types of comments. Backslash is not technically an escape character—it is passed on to LaTeX—but ⟨q|pic⟩ uses it for parsing; for example, `\#` is not treated as initiating a comment. For completeness, we simply list, in order, the steps ⟨q|pic⟩ takes to parse a line.

1. Split the line into *entities*. Typically, an entity is a character. When `\` is found, it is combined with the following character as a single entity. Also, text within dollar signs, braces, or double quotes (which may be nested) is combined into a single entity, even if it contains whitespace.

2. Discard the first entity equal to `#` and anything following it.

3. Split the line using the entity `%`. The second and third portions will be used as comments; all subsequent parsing applies only to the first portion.

4. Group the entities into *subwords* by splitting on whitespace, colons, and semicolons. (A colon or semicolon is considered its own subword; whitespace is ignored.) A *word* is a collection of colon-delimited subwords.

5. If any subword is a user-defined macro (see Section 3.9), replace the macro with its expansion.[8] If the macro has n arguments, the n preceding words are removed and then used in place of the arguments in the macro expansion.

6. If one of the words is `DEFINE`, then define the macro; the next word will hereafter expand to the rest of the line. Any words preceding `DEFINE` are arguments to the macro.

7. Split the line using the word `;`. Each portion will be processed as a different command, and all these commands will be grouped inside a `LB` and `LE`. Subsequent parsing applies separately to each command.

8. Pull out any attribute specifiers (`attribute=value` for attributes discussed in Section 3.3, or `qwire`, `cwire`, `owire`). Remember these so they can be passed to the appropriate gate or wire. (If attributes occur in an otherwise empty semicolon-delimited command, they are attached to the implicit `LB`.) Complain if a subword following a colon is not an attribute specifier.

9. Parse the remaining words as one of the commands in Section 3. First, check if the first word is a ⟨q|pic⟩ command. If not, assume the first word is a wire, and search for a word that is a valid ⟨q|pic⟩ gate. If there is none, interpret the line as a (controlled) Z or NOT (depending on targets).

References

[1] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit. quant-ph/0410184, 2004.

[2] Samuel A. Kutin, David Petrie Moulton, and Lawren M. Smithline. Computation at a distance. *Chicago Journal of Theoretical Computer Science*, 2007.

[8]Except that if the user-defined macro is immediately preceded by `DEFINE` then this is treated as a redefinition and the macro is not expanded.