Guide for the Quantum-Circuit Package

June 4, 2023

Mc-Zen

Abstract

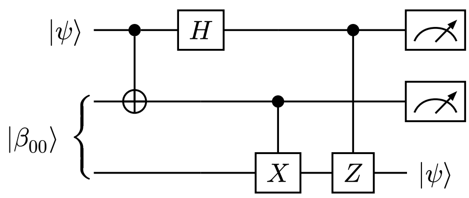
Quantum-Circuit is a library for creating quantum circuit diagrams in Typst.

Contents

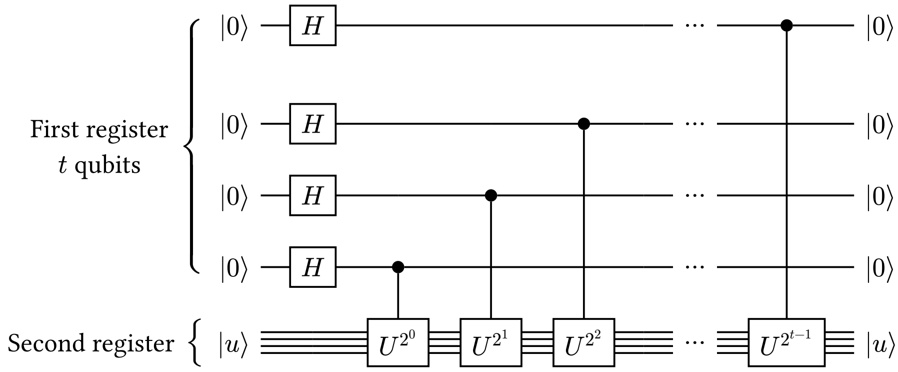
I Introduction

Section IV features a gallery of many gates that are possible to use with this library and how to create them. In Section VIII, you can find a variety of example figures along with the code.

Would you like to create quantum circuits directly in Typst? Maybe a circuit for quantum teleportation?



Or rather for phase estimation (the code for both examples can be found in Section VIII)?



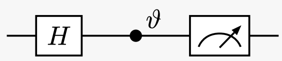
This library provides high-level functionality for generating these and more quantum circuit diagrams.

For those who work with the LaTeX packages `qcircuit` and `quantikz`, the syntax will be somewhat familiar. The wonderful thing about Typst is that the changes can be viewed instantaneously which makes it ever so much easier to design a beautiful quantum circuit. The syntax also has been updated a little bit to fit with concepts of the Typst language and many things like styling content is much simpler than with `quantikz` since it is directly supported in Typst.

II Basics

A basic circuit can be created by calling the `quantum-circuit()` command with a number of circuit elements:

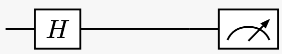
```
#quantum-circuit(
  1, gate($H$), phase($theta.alt$), meter(), 1
)
```



A quantum gate is created using the `gate()` command. Unlike `qcircuit` and `quantikz`, the math environment is not automatically entered for the content of the gate which allows to pass in any type of content (even images or tables). Use displaystyle math (for example `$ U_1 $` instead of `$U_1$`) to enable appropriate scaling of the gate for more complex mathematical expressions like double subscripts etc.

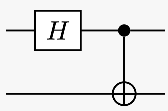
Consecutive gates are automatically joined with wires. Plain integers can be used to indicate a number of cells with just wire and no gate (where you would use a lot of `&`'s and `qw`'s in `quantikz`):

```
#quantum-circuit(
  1, gate($H$), 4, meter()
)
```



A new wire can be created by breaking the current wire with `[\ ]`:

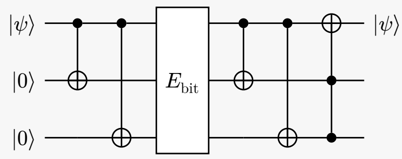
```
#quantum-circuit(
  1, gate($H$), control(1), 1, [\ ],
  2, targ(), 1
)
```



We can create a CX-gate by calling `control()` and passing the relative distance to the desired wire, e.g., `1` to the next wire, `2` to the second-next one or `-1` to the previous wire. Per default, the end of the vertical wire is just joined with the target wire without any decoration at all. Here, we make the gate a CX-gate by adding a `targ()` symbol on the second wire.

Let's look at a quantum bit-flipping error correction circuit. Here we encounter our first multi-qubit gate as well as wire labels:

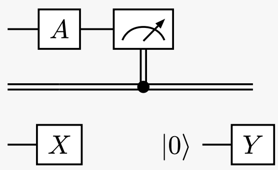
```
#quantum-circuit(
  lstick($|psi⟩$), control(1), control(2), mqgate($E_"bit"$, 3), control(1), control(2),
    targ(), rstick($|psi⟩$), [\ ],
  lstick($|0⟩$), targ(), 2, targ(), 1, control(-1), 1, [\ ],
  lstick($|0⟩$), 1, targ(), 2, targ(), control(-1), 1
)
```



Multi-qubit gates have a dedicated command `mqgate()` which takes the content as well as the number of qubits. Wires can be labelled at the beginning or the end with the `lstick()` and `rstick()` commands respectively.

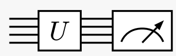
In many circuits, we need classical wires. This library generalizes the concept of quantumm classical and bundled wires and provides the `setwire()` command that allows all sorts of changes to the current wire setting. You may call `setwire()` with the number of wires to display:

```
#quantum-circuit(
  1, gate($A$), meter(target: 1), [\ ],
  setwire(2), 2, ctrl(), 2, [\ ],
  1, gate($X$), setwire(0), 1, lstick($|0⟩$),
    setwire(1), gate($Y$),
)
```



The `setwire` command produces no cells and can be called at any point on the wire. When a new wire is started, the default wire setting is restored automatically (quantum wire with default wire style, see Section III on how to customize the default). Calling `setwire(0)` removes the wire altogether until `setwire` is called with different arguments. More than two wires are possible and it lies in your hands to decide how many wires still look good. The distance between wires can also be specified:

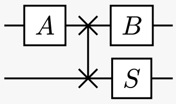
```
#quantum-circuit(
  setwire(4, wire-distance: 1.5pt), 1, gate($U$), meter()
)
```



III Circuit Styling

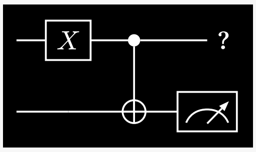
The `quantum-circuit()` command provides several options for styling the entire circuit. The parameters `row-spacing` and `column-spacing` allow changing the optical density of the circuit by adjusting the spacing between circuit elements vertically and horizontally.

```
#quantum-circuit(
  row-spacing: 5pt,
  column-spacing: 5pt,
  1, gate($A$), gate($B$), 1, [\ ],
  1, 1, gate($S$), 1
)
```



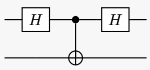
The `wire`, `color` and `fill` options provide means to customize line strokes and colors. This allows us to easily create "dark-mode" circuits:

```
#box(fill: black, quantum-circuit(
  wire: .7pt + white, // Wire and stroke color
  color: white,       // Default foreground and text color
  fill: black,        // Gate fill color
  1, gate($X$), control(1), rstick([*?*]), [\ ],
  1,1, targ(), meter(),
))
```



Furthermore, a common task is changing the total size of a circuit by scaling it up or down. Instead of tweaking all the parameters like `font-size`, `padding`, `row-spacing` etc. you can specify the `scale-factor` option which takes a percentage value:

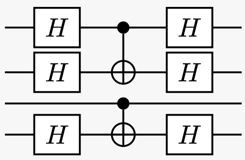
```
#quantum-circuit(
  scale-factor: 60%,
  1, gate($H$), control(1), gate($H$), 1, [\ ],
  1, 1, targ(), 2
)
```



Note, that this is different than calling Typst's builtin `scale()` function on the circuit which would scale it without affecting the layout, thus still reserving the same space as if unscaled!

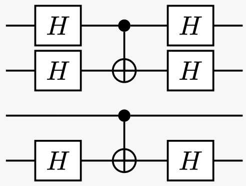
For an optimally layout, the height for each row is determined by the gates on that wire. For this reason, the wires can have different distances. To better see the effect, let's decrease the `row-spacing`:

```
#quantum-circuit(
    row-spacing: 2pt, min-row-height: 4pt,
    1, gate($H$), control(1), gate($H$), 1, [\ ],
    1, gate($H$), targ(), gate($H$), 1, [\ ],
    2, control(1), 2, [\ ],
    1, gate($H$), targ(), gate($H$), 1
)
```



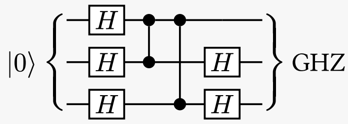
Setting the option `equal-row-heights` to `true` solves this problem (manually spacing the wires with lengths is still possible, see Section V):

```
#quantum-circuit(
    equal-row-heights: true,
    row-spacing: 2pt, min-row-height: 4pt,
    1, gate($H$), control(1), gate($H$), 1, [\ ],
    1, gate($H$), targ(), gate($H$), 1, [\ ],
    2, control(1), 2, [\ ],
    1, gate($H$), targ(), gate($H$), 1
)
```

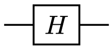
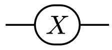
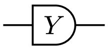
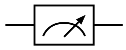
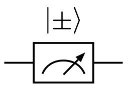
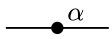
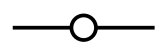
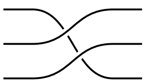
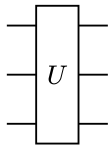
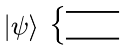
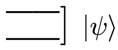
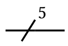
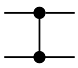
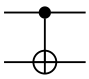
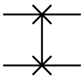
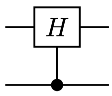
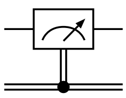


There is another option for `quantum-circuit()` that has a lot of impact on the looks of the diagram: `gate-padding`. This at the same time controls the default gate box padding and the distance of `lstick`'s and `rstick`'s to the wire. Need really wide or tight circuits?

```
#quantum-circuit(
    gate-padding: 2pt,
    row-spacing: 5pt, column-spacing: 7pt,
    lstick($|0〉$, num-qubits: 3), gate($H$), control(1),
      control(2), 1, rstick("GHZ", num-qubits: 3), [\ ],
    1, gate($H$), ctrl(), 1, gate($H$), 1, [\ ],
    1, gate($H$), 1, ctrl(), gate($H$), 1, [\ ],
)
```



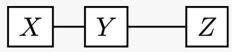
IV Gate Gallery

<table>
<tr><td>Normal gate</td><td></td><td>gate(H)</td><td>Round gate</td><td></td><td>gate(X, radius: 100%)</td></tr>
<tr><td>D gate</td><td></td><td>gate(Y, radius: (right: 100%))</td><td>Meter</td><td></td><td>meter()</td></tr>
<tr><td>Meter with label</td><td></td><td>meter(label: $lr(|±>)$)</td><td>Phase gate</td><td></td><td>phase($α$)</td></tr>
<tr><td>Control</td><td></td><td>ctrl()</td><td>Open control</td><td></td><td>ctrl(open: true)</td></tr>
<tr><td>Target</td><td></td><td>targ()</td><td>Swap target</td><td></td><td>targX()</td></tr>
<tr><td>Permutation gate</td><td></td><td>permute(2,0,1)</td><td>Multiqubit gate</td><td></td><td>mqgate(U, 3)</td></tr>
<tr><td>lstick</td><td>$|\psi\rangle$ —</td><td>lstick($|psi>$)</td><td>rstick</td><td>— $|\psi\rangle$</td><td>rstick($|psi>$)</td></tr>
<tr><td>Multi-qubit lstick</td><td></td><td>lstick($|psi>$, num-qubits: 2)</td><td>Multi-qubit rstick</td><td></td><td>rstick($|psi>$, num-qubits: 2, brace: "]")</td></tr>
<tr><td>midstick</td><td>— yeah —</td><td>midstick("yeah")</td><td>Wire bundle</td><td></td><td>nwire(5)</td></tr>
<tr><td>Controlled z-gate</td><td></td><td>control(1)
+
ctrl()</td><td>Controlled x-gate</td><td></td><td>control(1)
+
targ()</td></tr>
<tr><td>Swap gate</td><td></td><td>swap(1)
+
targX()</td><td>Controlled Hadamard</td><td></td><td>controlled(H, 1)
+
ctrl()</td></tr>
<tr><td>Meter to classical</td><td></td><td>meter(target: 1)
+
ctrl()</td><td></td><td></td><td></td></tr>
</table>

V Fine-Tuning

The `quantum-circuit()` command allows not only gates as well as content and string items but only length parameters which can be used to tweak the appearance of the circuit. Inserting a length value between gates adds a **horizontal space** of that length between the cells:

```
#quantum-circuit(
  gate($X$), gate($Y$), 10pt, gate($Z$)
)
```



In the background, this works like a grid gutter that is set to `0pt` by default. If a length value is inserted between the same two columns on different wires/rows, the maximum value is used for the space. In the same spirit, inserting multiple consecutive length values result in the largest being used, e.g., a `5pt, 10pt, 6pt` results in a `10pt` gutter in the corresponding position.

Putting a a length after the wire break item `[\ ]` produces a **vertical space** between the corresponding wires:

```
#quantum-circuit(
  gate($X$), [\ ], gate($Y$), [\ ], 10pt, gate($Z$)
)
```

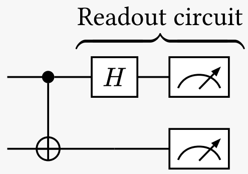
VI Annotations

Quantum-Circuit provides a way of making custom annotations through the `annotate()` interface. An `annotate()` object may be placed anywhere in the circuit, the position only matters for the draw order in case several annotations would overlap.

The `annotate()` command allows for querying cell coordinates of the circuit and passing in a custom draw function to draw globally in the circuit diagram.

Let's look at an example:

```
#quantum-circuit(
  1, control(1), gate($H$), meter(), [\ ],
  1, targ(), 1, meter(),
  annotate(0, (2, 4),
    (y, (x1, x2)) => {
      let brace = math.lr($#box(height: x2 - x1)}$)
      place(dx: x1, dy: y, rotate(brace, -90deg, origin: top))
      let content = [Readout circuit]
      style(styles => {
        let size = measure(content, styles)
        place(dx: x1 + (x2 - x1)/2 - size.width/2, dy: y - .6em - size.height,
content)
      })
  })
)
```

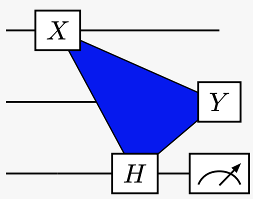


First, the call to `annotate()` asks for the y coordinate of the zeroth row (first wire) and the x coordinates of the second and forth column. The draw callback function then gets the corresponding coordinates as arguments and uses them to draw a brace and some text above the cells.

Note, that the circuit does not know how large the annotation is. If it goes beyond the circuits bounds, you may want to adjust the parameter `circuit-padding` of `quantum-circuit()` appropriately.

Another example, here we want to obtain coordinates for the cell centers. We can achieve this by adding 0.5 to the cell index. The fractional part of the number represents a percentage of the cell width/height.

```
#quantum-circuit(
  1, gate($X$), 2, [\ ],
  1, 2, gate($Y$), [\ ],
  1, 1, gate($H$), meter(),
  annotate((0.5, 1.5, 2.5), (1.5, 3.5, 2.5),
    ((y0, y1, y2), (x0, x1, x2)) => {
      path(
        (x0, y0), (x1, y1), (x2, y2),
        closed: true,
        fill: rgb("#1020EE50"), stroke: .5pt + black
      )
  })
)
```



VII Function Documentation

```
gate(
  content-type string ,
  arg: string none auto boolean
  a: integer symbol
  function float
)
```

This section contains a complete reference for every function in **quantum-circuit**.

Gates

- gate()
- mqgate()
- meter()
- permute()
- phantom()
- targ()
- ctrl()
- targX()
- phase()
- controlled()
- swap()
- control()

Decorations

- lstick()
- rstick()
- nwire()
- setwire()
- gategroup()
- slice()
- annotate()

Quantum Circuit

- quantum-circuit()

gate

Basic command for creating gates. Use this to create a simple gate, e.g. `gate($X$)`. For special gates, many other dedicated gate commands exist.

Note, that many of the parameters listed here are mostly used for derived gate function and do not need to be touched in all but very few cases.

Parameters

```
gate(
  content: content ,
  fill: none color ,
  radius: length dictionary ,
  box: boolean ,
  floating: boolean ,
  multi: dictionary ,
  size-hint: function ,
  draw-function: function ,
  gate-type,
  ..args
)
```

content `content`

What to show in the gate (none for special gates).

fill `none` or `color`

Gate backgrond fill color.

Default: `none`

radius `length` or `dictionary`

Gate rectangle border radius. Allows values like the builtin `rect()` function.

Default: `0pt`

box `boolean`

Whether this is a boxed gate (determines whether the outgoing wire will be drawn all through the gate (box: false) or not).

Default: `true`

floating `boolean`

Whether the content for this gate will be shown floating (i.e. no width is reserved).

Default: `false`

multi `dictionary`

Information for multi-qubit and controlled gates (see `mqgate()` .

Default: `none`

size-hint `function`

Size hint function.

Default: `default-size-hint`

draw-function `function`

Drawing function that produces the displayed content. Signature: (gate, draw-params).

- ..args (array): Optional additional arguments. When using a custom `draw-function` , you may store extra information here.

Default: `draw-boxed-gate`

mqgate

Basic command for creating multi-qubit or controlled gates. For the latter, you usually want to go for something like `control()`, `swap()`, or `controlled()`.

Parameters

```
mqgate(
  content: content,
  num-qubits: integer,
  fill: none color,
  radius: length dictionary,
  box: boolean,
  label: content,
  control: boolean,
  wire-count,
  extent: auto length,
  size-all-wires: none boolean,
  draw-function,
  ..args
)
```

num-qubits `integer`

Number of qubits (or relative wire count for controlled gates).

fill `none` or `color`

Gate backgrond fill color.

Default: `none`

radius `length` or `dictionary`

Gate rectangle border radius. Allows values like the builtin `rect()` function.

Default: `0pt`

box `boolean`

Whether this is a boxed gate (determines whether the outgoing wire will be drawn all through the gate (box: false) or not)

Default: `true`

label `content`

Optional label on the vertical wire.

Default: `none`

control `boolean`

If true, this gate draws a vertical control wire.

Default: `false`

extent `auto` or `length`

How much to extent the gate beyond the first and last wire, default is to make it align with an X gate (so [size of x gate] / 2).

Default: `auto`

size-all-wires `none` or `boolean`

A single-qubit gate affects the height of the row it's being put on. For multi-qubit gate there are different possible behaviours:

- Affect size on only the first and last qubit (`false`)
- Affect the size of all qubits (`true`)
- Affect the size on no qubit (`none`)

Default: `false`

meter

Draw a meter box representing a measurement.

Parameters

```
meter(
  target: none integer,
  wire-count: integer,
  label: content
)
```

target `none` or `integer`

Draw a control wire to the given target qubit relative to this qubit if not `none`.

Default: `none`

wire-count `integer`

Wire count for the control wire.

Default: `2`

label `content`

Label to show above the meter.

Default: `none`

permute

Create a viualized permutation gate which maps the qubits $q_k, q_{k+1}, \ldots$ to the qubits $q_{p(k)}, q_{p(k+1)}, \ldots$ when placed on qubit k. The permutation map is given by the `qubits` argument.

Example:

`permute(1, 0)` when placed on the second wire swaps the second and third wire.

`permute(2, 0, 1)` when placed on wire 0 maps $(0, 1, 2) \mapsto (2, 0, 1)$

Note, that the wiring is not very sophisticated and will probably look best for relatively simple permutations. Furthermore, it only works with quantum wires.

- ..qubits (array): Qubit permutation specification.

Parameters

```
permute(
  ..qubits,
  width: length
)
```

width `length`

Width of the permutation gate.

Default: `30pt`

phantom

Create an invisible (phantom) gate for reserving space. If `content` is provided, the `height` and `width` parameters are ignored and the gate will take the size it would have if `gate(content)` was called.

Instead specifying width and/or height will create a gate with exactly the given size (without padding).

Parameters

```
phantom(
  content: content,
  width: length,
  height: length
)
```

content `content`

Content to measure for the phantom gate size.

Default: `none`

width `length`

Width of the phantom gate (ignored if `content` is not `none`).

Default: `0pt`

height `length`

Height of the phantom gate (ignored if `content` is not `none`).

Default: `0pt`

targ

Target element for controlled x operations (⊕).

Parameters

```
targ(
  fill: none color boolean,
  size: length
)
```

fill `none` or `color` or `boolean`

Fill color for the target circle. If set to `true`, the target is filled with the circuits background color.

Default: `none`

size `length`

Size of the target symbol.

Default: `4.3pt`

ctrl

Target element for controlled z operations (•).

Parameters

```
ctrl(
  open: boolean,
  fill: none color,
  size: length
)
```

open `boolean`

Whether to draw an open dot.

Default: `false`

fill `none` or `color`

Fill color for the circle or stroke color if `open: true`.

Default: `none`

size `length`

Size of the control circle.

Default: `2.3pt`

targX

Target element for SWAP operations (×) without vertical wire).

Parameters

```
targX(size: length)
```

size `length`

Size of the target symbol.

Default: `7pt`

phase

Create a phase gate shown as a point on the wire together with a label.

Parameters

```
phase(
  label: content,
  open: boolean,
  fill: none color,
  size: length
)
```

label `content`

Angle value to display.

open `boolean`

Whether to draw an open dot.

Default: `false`

fill `none` or `color`

Fill color for the circle or stroke color if `open: true`.

Default: `none`

size `length`

Size of the circle.

Default: `2.3pt`

lstick

Basic command for labelling a wire at the start.

Parameters

```
lstick(
  label: content,
  num-qubits: content,
  brace: auto none string
)
```

label `content`

Label to display, e.g. `$|0⟩$`.

num-qubits `content`

How many qubits the `lstick` should span.

Default: `1`

brace `auto` or `none` or `string`

If `brace` is `auto`, then a default `{` brace is shown only if `num-qubits > 1`. A brace is always shown when explicitly given, e.g., `"}"`, `"["` or `"|"`. No brace is shown for `brace: none`.

Default: `auto`

rstick

Basic command for labelling a wire at the end.

Parameters

```
rstick(
  label: content,
  num-qubits: content,
  brace: auto none string
)
```

label `content`

Label to display, e.g. `$|0⟩$`.

num-qubits `content`

How many qubits the `rstick` should span.

Default: `1`

brace `auto` or `none` or `string`

If `brace` is `auto`, then a default `}` brace is shown only if `num-qubits > 1`. A brace is always shown when explicitly given, e.g., `"}"`, `"["` or `"|"`. No brace is shown for `brace: none`.

Default: `auto`

nwire

Creates a symbol similar to `\qwbundle` on `quantikz`. Annotates a wire to be a bundle of quantum or classical wires.

- n(integer):

Parameters

```
nwire(n)
```

controlled

Create a controlled gate. See also `control()`. This function however may be used to create controlled gates where a gate box is at both ends of the control wire.

Example: `controlled($H$, 2)`.

Parameters

```
controlled(
  content: content,
  relative-qubit: integer,
  wire-count: integer,
  draw-function,
  ..args
)
```

content `content`

Gate content to display.

relative-qubit `integer`

Target qubit relative to this one.

wire-count `integer`

Wire count for the control wire.

- draw-function (function). See `gate()`.
- ..args (array): Optional, additional arguments to be stored in the gate.

Default: `1`

swap

Creates a SWAP operation with another qubit.

Parameters

```
swap(
  relative-qubit: integer,
  size: length
)
```

relative-qubit `integer`

Target qubit relative to this one.

size `length`

Size of the target symbol.

Default: `7pt`

control

Creates a control with a vertical wire to another qubit.

Parameters

```
control(
  relative-qubit: integer,
  wire-count: integer,
  open: boolean,
  fill: none color,
  size: length
)
```

relative-qubit `integer`

Target qubit relative to this one.

wire-count `integer`

Wire count for the control wire.

Default: `1`

open `boolean`

Whether to draw an open dot.

Default: `false`

fill `none` or `color`

Fill color for the circle or stroke color if `open: true`.

Default: `none`

size `length`

Size of the control circle.

Default: `2.3pt`

setwire

Set current wire mode (0: none, 1 wire: quantum, 2 wires: classical, more are possible) and optionally the stroke style.

The wire style is reset for each row.

Parameters

```
setwire(
  wire-count: integer,
  stroke: none stroke,
  wire-distance: length
)
```

wire-count `integer`

Number of wires to display.

stroke `none` or `stroke`

When given, the stroke is applied to the wire. Otherwise the current stroke is kept.

Default: `none`

wire-distance `length`

Distance between wires.

Default: `1pt`

gategroup

Highlight a group of circuit elements by drawing a rectangular box around them.

Parameters

```
gategroup(
  wires: integer ,
  steps: integer ,
  padding: length dictionary ,
  stroke: stroke ,
  fill: color ,
  radius: length dictionary
)
```

wires `integer`

Number of wires to include.

steps `integer`

Number of columns to include.

padding `length` or `dictionary`

Padding of rectangle. May be one length for all sides or a dictionary with the keys `left` , `right` , `top` , `bottom` and `default` . Not all keys need to be specified. The value for `default` is used for the omitted sides or `0pt` if no `default` is given.

Default: `0pt`

stroke `stroke`

Stroke for rectangle.

Default: `.7pt`

fill `color`

Fill color for rectangle.

Default: `none`

radius `length` or `dictionary`

Corner radius for rectangle.

Default: `0pt`

slice

Slice the circuit vertically, showing a separation line between columns.

Parameters

```
slice(
  wires: integer ,
  label: content ,
  stroke: stroke ,
  dx,
  dy
)
```

wires `integer`

Number of wires to slice.

Default: `0`

label `content`

Label for the slice.

Default: `none`

stroke `stroke`

Line style for the slice.

Default: `(paint: red, thickness: .7pt, dash: "dashed")`

annotate

Lower-level interface to the cell coordinates to create an arbitrary annotatation by passing a custom function.

This function is passed the coordinates of the specified cell rows and columns.

Parameters

```
annotate(
  rows: integer array ,
  columns: integer array ,
  callback: function
)
```

rows `integer` or `array`

Row indices for which to obtain coordinates.

columns `integer` or `array`

Column indices for which to obtain coordinates.

callback `function`

Function to call with the obtained coordinates. The signature should be with signature `(row-coords, col-coords) => {}`. This function is expected to display the content to draw in absolute coordinates within the circuit.

quantum-circuit

Create a quantum circuit diagram. Content items may be

- Gates created by one of the many gate commands (`gate()`, `mqgate()`, `meter()`, …)
- `[\ ]` for creating a new wire/row
- Commands like `setwire()` or `gategroup()`
- Integers for creating cells filled with the current wire setting
- Lengths for creating space between rows or columns
- Plain content or strings to be placed on the wire
- `lstick`, `midstick` or `rstick` for placement next to the wire

Parameters

```
quantum-circuit(
  wire: stroke,
  row-spacing: length,
  column-spacing: length,
  min-row-height: length,
  min-column-width: length,
  gate-padding: length,
  equal-row-heights: boolean,
  color: color,
  fill: color,
  font-size: length,
  scale-factor: relative length,
  baseline: length content string,
  circuit-padding: dictionary,
  ..content
)
```

wire `stroke`

Style for drawing the circuit wires. This can take anything that is valid for the stroke of the builtin `line` function.

Default: `.7pt + black`

row-spacing `length`

Spacing between rows.

Default: `12pt`

column-spacing `length`

Spacing between columns.

Default: `12pt`

min-row-height `length`

Minimum height of a row (e.g. when no gates are given).

Default: `10pt`

min-column-width `length`

Minimum width of a column.

Default: `10pt`

gate-padding `length`

General padding setting including the inset for gate boxes and the distance of `lstick` and co. to the wire.

Default: `.4em`

equal-row-heights `boolean`

If true, then all rows will have the same height and the wires will have equal distances orienting on the highest row.

Default: `false`

color `color`

Foreground color, default for strokes, text, controls etc. If you want to have dark-themed circuits, set this to white for instance and update `wire` and `fill` accordingly.

Default: `black`

fill `color`

Default fill color for gates.

Default: `white`

font-size `length`

Default font size for text in the circuit.

Default: `10pt`

scale-factor `relative length`

Total scale factor applied to the entire circuit without changing proportions

Default: `100%`

baseline `length` or `content` or `string`

Set the baseline for the circuit. If a content or a string is given, the baseline will be adjusted auto- matically to align with the center of it. One useful application is `"="` so the circuit aligns with the equals symbol.

Default: `0pt`

circuit-padding `dictionary`

Padding for the circuit (e.g., to accomodate for annotations) in form of a dictionary with possible keys `left`, `right`, `top` and `bottom`. Not all of those need to be specified.

This setting basically just changes the size of the bounding box for the circuit and can be used to increase it when labels or annotations extend beyond the actual circuit.

- ..content (array): Items, gates and circuit commands (see description).

Default: `none`

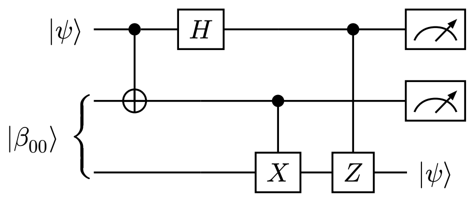
VIII Demo

This section demonstrates the use of the **quantum-circuit** library by reproducing some figures from the famous book *Quantum Computation and Quantum Information* by Nielsen and Chuang [1].

VIII.a Quantum teleportation

Quantum teleportation circuit reproducing the Figure 4.15 in [1].

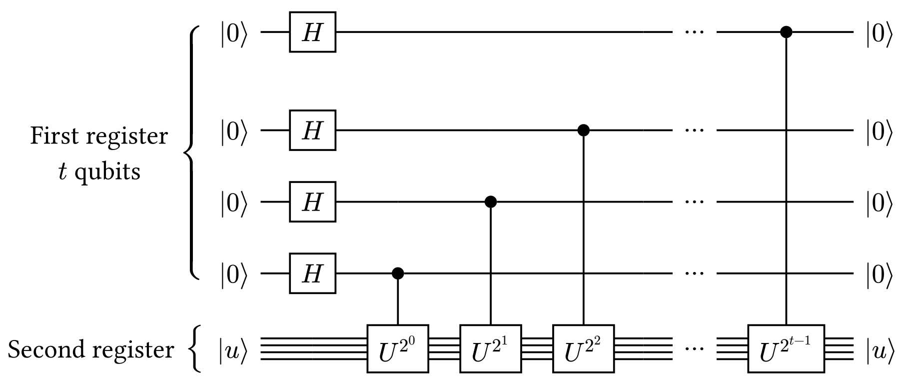
```
#quantum-circuit(
  lstick($|psi⟩$),  control(1), gate($H$), 1, ctrl(), meter(), [\ ],
  lstick($|beta_00⟩$, num-qubits: 2), targ(), 1, ctrl(), 1, meter(), [\ ],
  3, controlled($X$, -1), controlled($Z$, -2),  midstick($|psi⟩$)
)
```



VIII.b Quantum phase estimation

Quantum phase estimation circuit reproducing the Figure 5.2 in [1].

```
#quantum-circuit(
  setwire(0), lstick(align(center)[First register\ $t$ qubits], num-qubits: 4), lstick($|0⟩$),
    setwire(1), gate($H$), 4, midstick($ dots $), control(4), rstick($|0⟩$), [\ ], 10pt,
  setwire(0), phantom(width: 13pt), lstick($|0⟩$), setwire(1), gate($H$), 2, control(3), 1,
    midstick($ dots $), 1, rstick($|0⟩$), [\ ],
  setwire(0), 1, lstick($|0⟩$), setwire(1), gate($H$), 1, control(2), 2,
    midstick($ dots $), 1, rstick($|0⟩$), [\ ],
  setwire(0), 1, lstick($|0⟩$), setwire(1), gate($H$), control(1), 3, midstick($ dots $), 1,
    rstick($|0⟩$), [\ ],
  setwire(0), lstick([Second register], num-qubits: 1, brace: "{"), lstick($|u⟩$),
    setwire(4, wire-distance: 1.3pt), 1, gate($ U^2^0 $, -1), gate($ U^2^1 $, -2), gate($ U^2^2 $, -3),
    1, midstick($ dots $), gate($ U^2^(t-1) $, -5), rstick($|u⟩$)
)
```



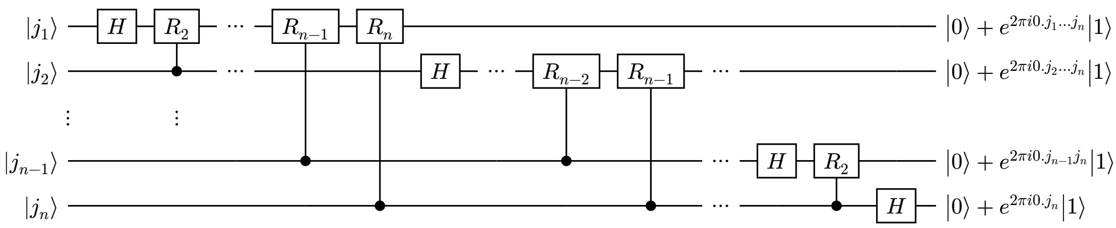
VIII.c Quantum Fourier transform:

Circuit for performing the quantum Fourier transform, reproducing the Figure 5.1 in [1].

```
#quantum-circuit(
  scale-factor: 85%,
  row-spacing: 5pt,
  column-spacing: 8pt,
  lstick($|j_1⟩$), gate($H$), controlled($R_2$, 1), midstick($ dots $),
    controlled($R_(n-1)$, 3), controlled($R_n$, 4), 8, rstick($|0⟩+e^(2pi i 0.j_1 dots j_n)|1⟩$),[\ ],
  lstick($|j_2⟩$), 1, ctrl(), midstick($ dots $), 2, gate($H$), midstick($ dots $),
    controlled($R_(n-2)$, 2), controlled($R_(n-1)$, 3), midstick($ dots $), 3,
    rstick($|0⟩+e^(2pi i 0.j_2 dots j_n)|1⟩$), [\ ],

  setwire(0),  midstick($dots.v$), 1, midstick($dots.v$), [\ ],

  lstick($|j_(n-1)⟩$), 3, ctrl(), 3, ctrl(), 1, midstick($ dots $), gate($H$), controlled($R_(2)$, 1),
    1, rstick($|0⟩+e^(2pi i 0.j_(n-1)j_n)|1⟩$), [\ ],
  lstick($|j_n⟩$), 4, ctrl(), 3, ctrl(), midstick($ dots $), 1, ctrl(), gate($H$),
    rstick($|0⟩+e^(2pi i 0.j_n)|1⟩$)
)
```

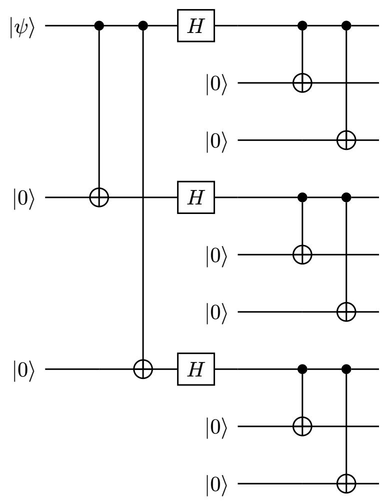


VIII.d Shor Nine Qubit Code

Encoding circuit for the Shor nine qubit code. This diagram repdoduces Figure 10.4 in [1]

```
#let ancillas = (setwire(0), 5, lstick($|0⟩$), setwire(1), targ(), 2,
[\ ],
setwire(0), 5, lstick($|0⟩$), setwire(1), 1, targ(), 1, [\ ])

#quantum-circuit(
  scale-factor: 80%,
  lstick($|ψ⟩$), 1, 10pt, control(3), control(6), gate($H$), 1, 15pt,
    control(1), control(2), 1, [\ ],
  ..ancillas,
  lstick($|0⟩$), 1, targ(), 1, gate($H$), 1, control(1), control(2),
    1, [\ ],
  ..ancillas,
  lstick($|0⟩$), 2, targ(),  gate($H$), 1, control(1), control(2),
    1, [\ ],
  ..ancillas
)
```

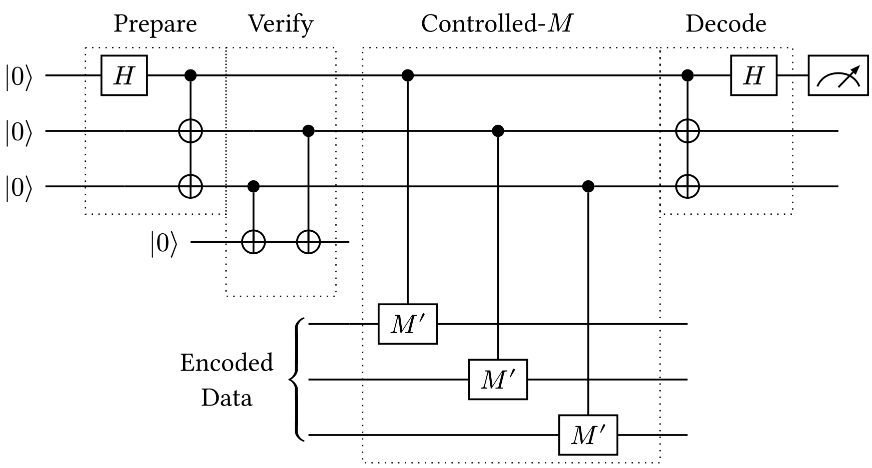


VIII.e Fault-Tolerant Measurement

Circuit for performing the quantum Fourier transform, reproducing the Figure 10.28 in [1].

```
#let mark(text, col1, col2) = annotate(0, (col1, col2),
  (y, (x1, x2)) => style(styles => {
    let size = measure(text, styles)
    place(dx: x1 + (x2 - x1)/2 - size.width/2, dy: y - .6em - size.height, text)
  })
)
#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

#quantum-circuit(
  row-spacing: 6pt,
  circuit-padding: (top: 2em),
  lstick($|0〉$), 10pt, group(3, 2), gate($H$), control(2), 3pt, group(4, 2), 3, group(7, 3),
    control(4), 2, 10pt, group(3, 2), control(2), gate($H$), meter(), [\ ],
  lstick($|0〉$), 1, targ(), 1, control(2), 2, control(4), 1, targ(), 2, [\ ],
  lstick($|0〉$), 1, targ(), control(1), 4, control(4), targ(), 2, [\ ],
  setwire(0), 2, lstick($|0〉$), setwire(1), targ(), targ(), 1, [\ ], 10pt,
  setwire(0), 4, lstick(align(center)[Encoded\ Data], num-qubits: 3), setwire(1), 1,
    gate($M'$), 3, [\ ],
  setwire(0), 5,  setwire(1), 2, gate($M'$), 2, [\ ],
  setwire(0), 5,  setwire(1), 3, gate($M'$), 1,
  mark("Prepare", 1, 3),
  mark("Verify", 3, 5),
  mark([Controlled-$M$], 5, 9),
  mark("Decode", 9, 11)
)
```

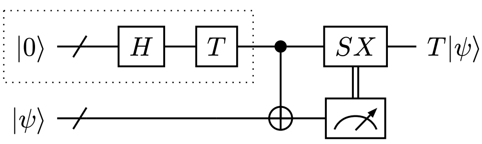


VIII.f Fault-Tolerant Gate Construction

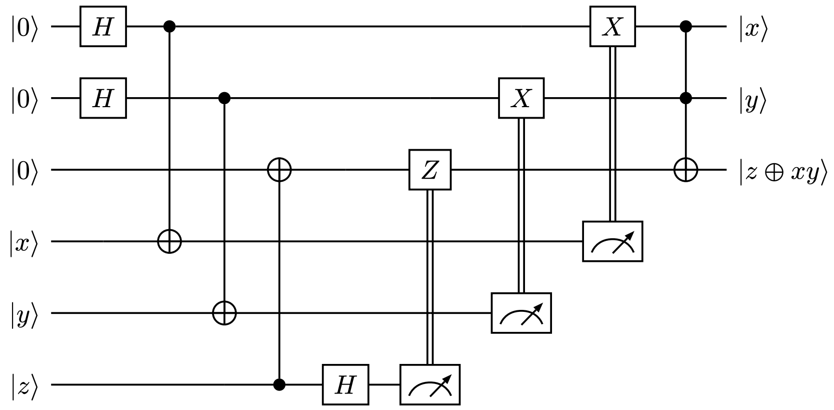
The following two circuits reproduce figures from Exercise 10.66 and 10.68 on construction fault-tolerant $\frac{\pi}{8}$ and Toffoli gates in [1].

```
#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

#quantum-circuit(
  group(1, 4, padding: (left: 1.5em)), lstick($|0〉$), nwire(""), gate($H$), gate($T$),
    control(1), gate($S X$), rstick($T|ψ〉$), [\ ],
  lstick($|ψ〉$), nwire(""), 2, targ(), meter(target: -1),
)
```

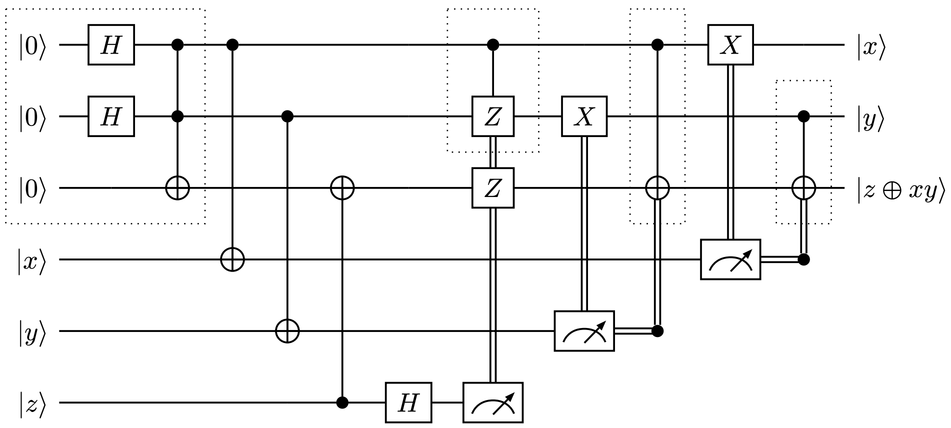


```
#quantum-circuit(
  lstick($|0⟩$), gate($H$), control(3), 5, gate($X$), control(2), rstick($|x⟩$), [\ ],
  lstick($|0⟩$), gate($H$), 1, control(3), 3, gate($X$), 1, ctrl(), rstick($|y⟩$), [\ ],
  lstick($|0⟩$), 3, targ(), 1, gate($Z$), 2, targ(), rstick($|z plus.circle x y⟩$), [\ ],
  lstick($|x⟩$), 1, targ(), 5, meter(target: -3), [\ ],
  lstick($|y⟩$), 2, targ(), 3, meter(target: -3), [\ ],
  lstick($|z⟩$), 3, control(-3), gate($H$), meter(target: -3)
)
```




```
#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

#quantum-circuit(
  group(3, 3, padding: (left: 1.5em)), lstick($|0⟩$), gate($H$), control(2), control(3), 3,
    group(2, 1),control(1), 1, group(3, 1), control(2), gate($X$), 1, rstick($|x⟩$), [\ ],
  lstick($|0⟩$), gate($H$), ctrl(), 1, control(3), 2, gate($Z$), gate($X$), 2, group(2, 1),
    control(1), rstick($|y⟩$), [\ ],
  lstick($|0⟩$), 1, targ(), 2, targ(), 1, gate($Z$), 1, targ(fill: true), 1, targ(fill: true),
    rstick($|z plus.circle x y⟩$), [\ ],
  lstick($|x⟩$), 2, targ(), 6, meter(target: -3), setwire(2), control(-1, wire-count: 2), [\ ],
  lstick($|y⟩$), 3, targ(), 3, meter(target: -3), setwire(2), control(-2, wire-count: 2), [\ ],
  lstick($|z⟩$), 4, control(-3), gate($H$), meter(target: -4)
)
```



Bibliography

[1] M. A. Nielsen, and I. L. Chuang, *Quantum Computation and Quantum Information*, 2nd ed., Cambridge Cambridge University Press, 2022.