$$A \xrightarrow{f} B$$

fletcher

(noun) a maker of arrows

A Typst package for diagrams with lots of arrows, built on top of CeTZ.

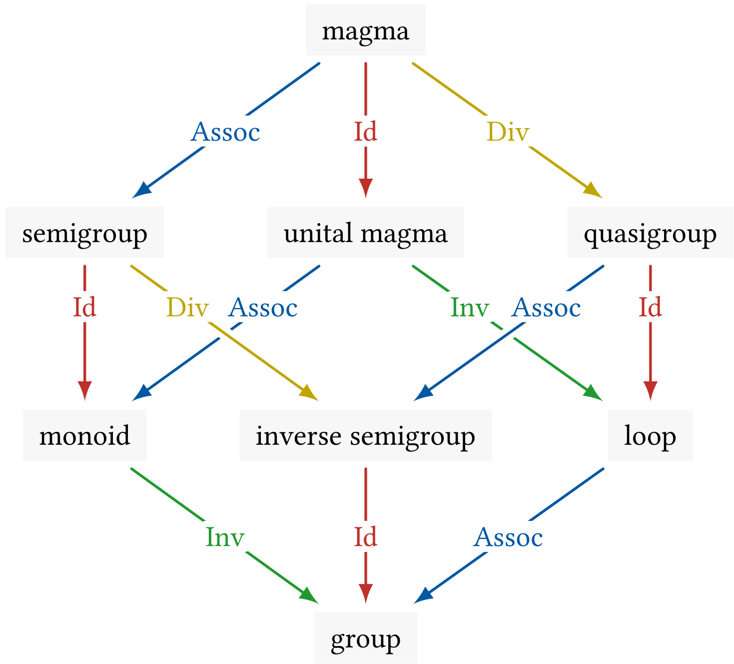
Commutative diagrams, flow charts, state machines, block diagrams...

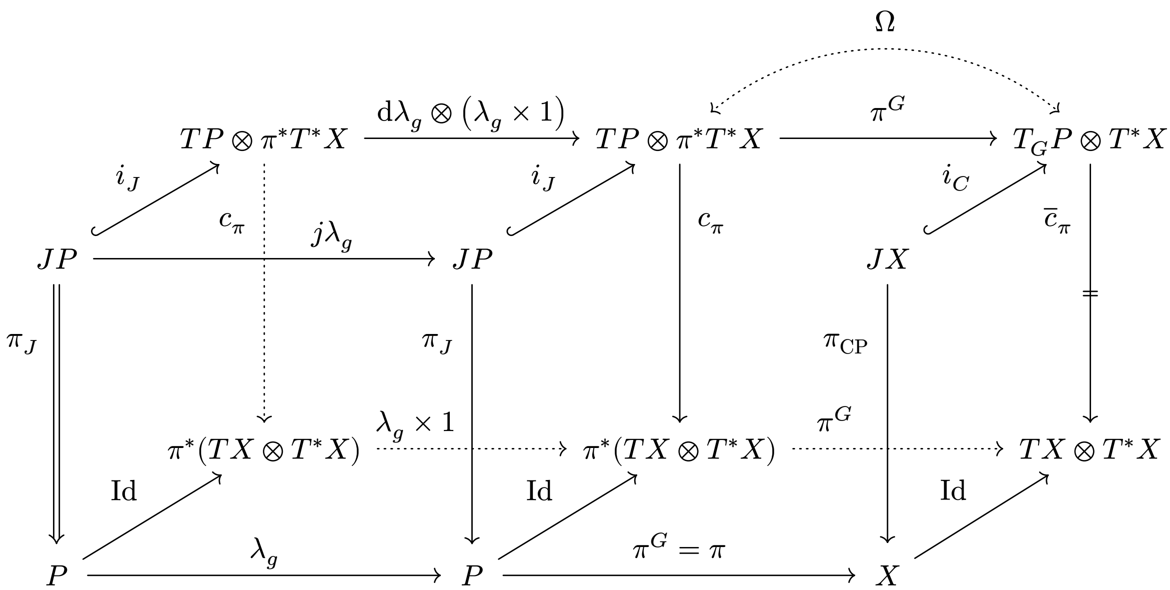
`github.com/Jollywatt/typst-fletcher`

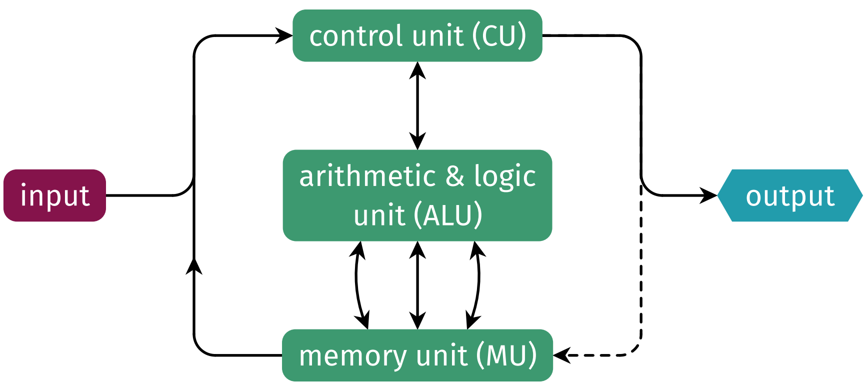
Version 0.4.5

Guide

Reference


magma
Assoc
Id
Div
semigroup
unital magma
quasigroup
Id
Div
Assoc
Inv
Assoc
Id
monoid
inverse semigroup
loop
Inv
Id
Assoc
group


Ω
$TP \otimes \pi^* T^* X$
$\mathrm{d}\lambda_g \otimes (\lambda_g \times 1)$
$TP \otimes \pi^* T^* X$
π^G
$T_G P \otimes T^* X$
i_J
i_J
i_C
c_π
$j\lambda_g$
c_π
$\bar{c}_\pi$
JP
JP
JX
π_J
π_J
π_{CP}
$\lambda_g \times 1$
π^G
$\pi^*(TX \otimes T^* X)$
$\pi^*(TX \otimes T^* X)$
$TX \otimes T^* X$
Id
Id
Id
λ_g
$\pi^G = \pi$
P
P
X

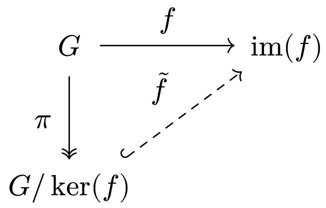

control unit (CU)
input
arithmetic & logic unit (ALU)
output
memory unit (MU)

Usage examples

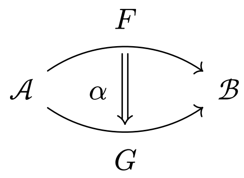
Avoid importing everything with * as many internal functions are also exported.

```
#import "@preview/fletcher:0.4.5" as fletcher: diagram, node, edge
```

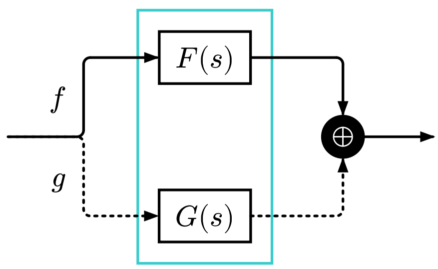
```
// You can specify nodes in math-mode, separated by `&`:
#diagram($
  G edge(f, ->) edge("d", pi, ->>) & im(f) \
  G slash ker(f) edge("ur", tilde(f), "hook-->")
$)
```




```
// Or you can use code-mode, with variables, loops, etc:
#diagram(spacing: 2cm, {
  let (A, B) = ((0,0), (1,0))
  node(A, $cal(A)$)
  node(B, $cal(B)$)
  edge(A, B, $F$, "->", bend: +35deg)
  edge(A, B, $G$, "->", bend: -35deg)
  let h = 0.2
  edge((.5,-h), (.5,+h), $alpha$, "=>")
})
```




```
#diagram(
  spacing: (10mm, 5mm), // wide columns, narrow rows
  node-stroke: 1pt,     // outline node shapes
  edge-stroke: 1pt,     // make lines thicker
  mark-scale: 60%,      // make arrowheads smaller
  edge((-2,0), "r,u,r", "-|>", $f$, label-side: left),
  edge((-2,0), "r,d,r", "..|>", $g$),
  node((0,-1), $F(s)$),
  node((0,+1), $G(s)$),
  node(enclose: ((0,-1), (0,+1)), stroke: teal, inset: 8pt),
  edge((0,+1), (1,0), "..|>", corner: left),
  edge((0,-1), (1,0), "-|>", corner: right),
 node((1,0), text(white, $ plus.circle $), inset: 2pt, fill: black),
  edge("-|>"),
)
```

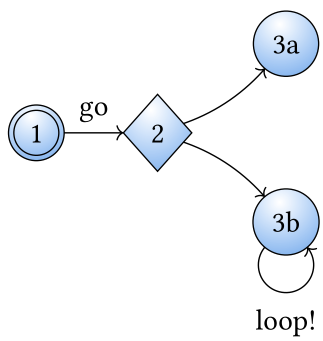



```
An equation $f: A -> B$ and \
an inline diagram #diagram(
  node-inset: 2pt,
  label-sep: 0pt,
  $A edge(->, text(#0.8em, f)) & B$
).
```

An equation $f : A \to B$ and

an inline diagram $A \xrightarrow{f} B$.

```
#import fletcher.shapes: diamond
#diagram(
  node-stroke: black + 0.5pt,
  node-fill: gradient.radial(white, blue, center: (40%, 20%),
                             radius: 150%),
  spacing: (10mm, 5mm),
  node((0,0), [1], name: <1>, extrude: (0, -4)), // double stroke
  node((1,0), [2], name: <2>, shape: diamond),
  node((2,-1), [3a], name: <3a>),
  node((2,+1), [3b], name: <3b>),
  edge(<1>, <2>, [go], "->"),
  edge(<2>, <3a>, "->", bend: -15deg),
  edge(<2>, <3b>, "->", bend: +15deg),
  edge(<3b>, <3b>, "->", bend: -130deg, label: [loop!]),
)
```



Diagrams

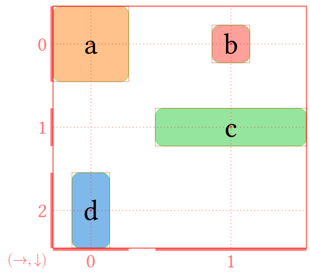
Diagrams created with `diagram()` are a collection of *nodes* and *edges* rendered on a CeTZ canvas.

Elastic coordinates

Diagrams are laid out on a *flexible coordinate grid*, visible when the `debug` option of `diagram()` is turned on. When a node is placed, the rows and columns grow to accommodate the node's size, like a table.

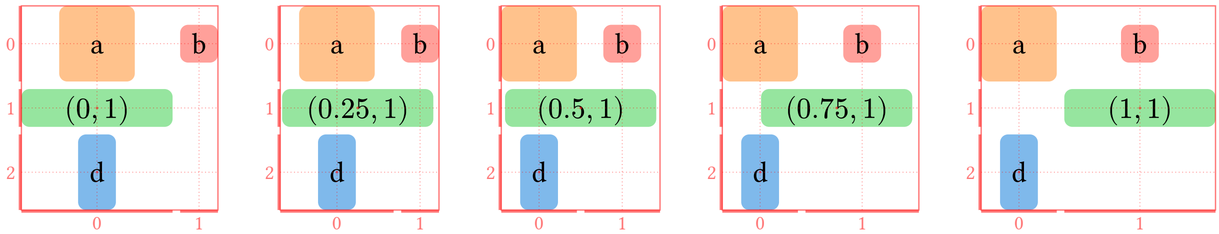
By default, coordinates (u, v) have u going $\rightarrow$ and v going $\downarrow$. This can be changed with the `axes` option of `diagram()`. The `cell-size` option is the minimum row and column width, and `spacing` is the gutter between rows and columns.

```
#let c = (orange, red, green, blue).map(x => x.lighten(50%))
#diagram(
  debug: 2,
  spacing: 10pt,
  node-corner-radius: 3pt,
  node((0,0), [a], fill: c.at(0), width: 10mm, height: 10mm),
  node((1,0), [b], fill: c.at(1), width:  5mm, height:  5mm),
  node((1,1), [c], fill: c.at(2), width: 20mm, height:  5mm),
  node((0,2), [d], fill: c.at(3), width:  5mm, height: 10mm),
)
```



Fractional coordinates

So far, this is just like a table — however, coordinates can be *fractional*. These are dealt with by linearly interpolating the diagram between what it would be if the coordinates were rounded up or down.

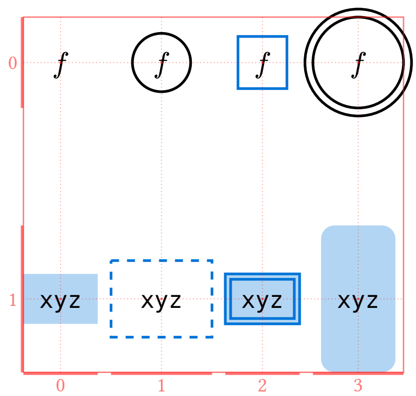


Nodes

`node((x, y), label, ..options)`

Nodes are content centered at a particular coordinate. They can be circular, rectangular, or any custom shape. Nodes automatically fit to the size of their label (with an `inset`), but can also be given an exact `width`, `height`, or `radius`, as well as a `stroke` and `fill`. For example:

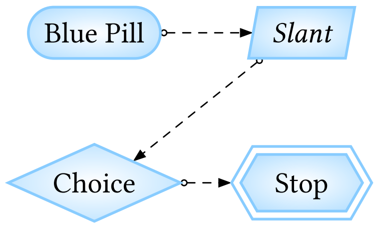
```
#diagram(
  debug: true, // show a coordinate grid
  spacing: (5pt, 4em), // small column gaps, large row spacing
  node((0,0), $f$),
  node((1,0), $f$, stroke: 1pt),
  node((2,0), $f$, stroke: blue, shape: rect),
  node((3,0), $f$, stroke: 1pt, radius: 6mm, extrude: (0, 3)),
  {
    let b = blue.lighten(70%)
    node((0,1), `xyz`, fill: b, )
    let dash = (paint: blue, dash: "dashed")
    node((1,1), `xyz`, stroke: dash, inset: 1em)
    node((2,1), `xyz`, fill: b, stroke: blue, extrude: (0, -2))
    node((3,1), `xyz`, fill: b, height: 5em, corner-radius: 5pt)
  }
)
```



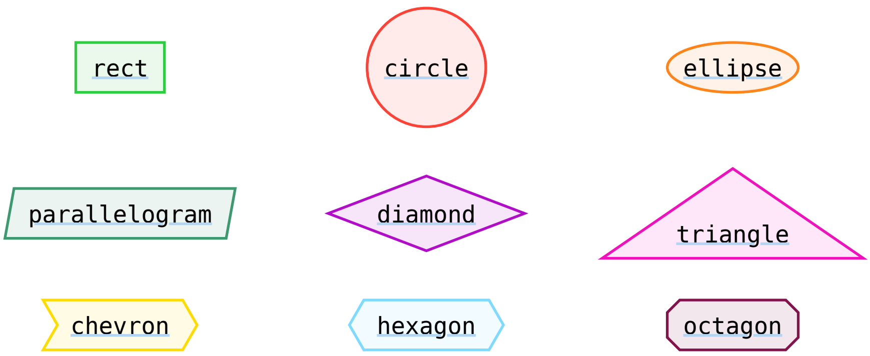
Node shapes

By default, nodes are circular or rectangular depending on the aspect ratio of their label. The `shape` option accepts `rect`, `circle`, various shapes provided in the `fletcher.shapes` submodule, or a function.

```
#import fletcher.shapes: pill, parallelogram, diamond, hexagon
#let theme = rgb("8cf")
#diagram(
  node-fill: gradient.radial(white, theme, radius: 100%),
  node-stroke: theme,
  (
    node((0,0), [Blue Pill], shape: pill),
    node((1,0), [_Slant_], shape: parallelogram.with(angle: 20deg)),
    node((0,1), [Choice], shape: diamond),
    node((1,1), [Stop], shape: hexagon, extrude: (-3, 0), inset: 10pt),
  ).intersperse(edge("o--|>")).join()
)
```



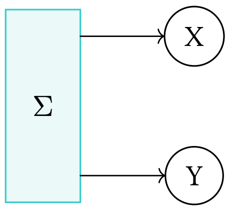
Custom node shapes may be implemented with CeTZ via the `shape` option of `node()`, but it is up to the user to support outline extrusion for custom shapes.



Node groups

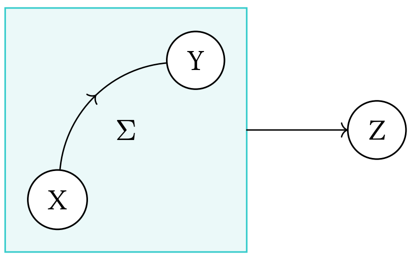
Nodes are usually centered at a particular coordinate, but they can also `enclose` multiple centers. When the `enclose` option of `node()` is given, the node automatically resizes.

```
#diagram(
  node-stroke: 0.6pt,
  node($Sigma$, enclose: ((1,1), (1,2)), // a node spanning multiple centers
      inset: 10pt, stroke: teal, fill: teal.lighten(90%), name: <bar>),
  node((2,1), [X]),
  node((2,2), [Y]),
  edge((1,1), "r", "->", snap-to: (<bar>, auto)),
  edge((1,2), "r", "->", snap-to: (<bar>, auto)),
)
```



You can also `enclose` other nodes by coordinate or `name` to create node groups:

```
#diagram(
  node-stroke: 0.6pt,
  node-fill: white,
  node((0,1), [X]),
  edge("->-", bend: 40deg),
  node((1,0), [Y], name: <y>),
  node($Sigma$, enclose: ((0,1), <y>), inset: 10pt,
      stroke: teal, fill: teal.lighten(90%), name: <group>),
  node((2.5,0.5), [Z], name: <z>),
  edge(<group>, <z>, "->"),
)
```

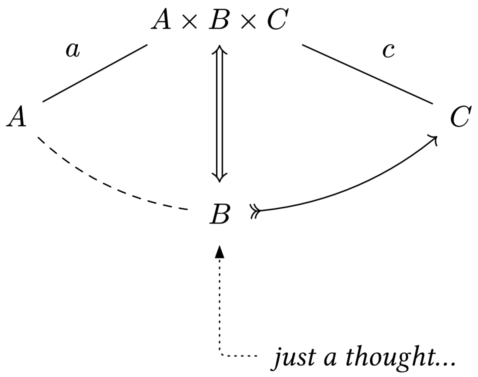


Edges

`edge(from, to, label, marks, ..options)`

Edges connect two coordinates. If there is a node at an endpoint, the edge attaches to the nodes' bounding shape (after applying the node's `outset`). An edges can have a `label`, can `bend` into an arc, and can have various arrow `marks`.

```
#diagram(spacing: (12mm, 6mm), {
  let (a, b, c, abc) = ((-1,0), (0,1), (1,0), (0,-1))
  node(abc, $A times B times C$)
  node(a, $A$)
  node(b, $B$)
  node(c, $C$)

  edge(a, b, bend: -18deg, "dashed")
  edge(c, b, bend: +18deg, "<-<<")
  edge(a, abc, $a$)
  edge(b, abc, "<=>")
  edge(c, abc, $c$)

  node((.6,3), [_just a thought..._])
  edge(b, "..|>", corner: right)
})
```



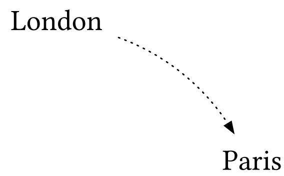
Specifying edge vertices

Generally, the first few arguments to `edge()` specify its `vertices`.

Implicit coordinates

To specify the start and end points of an edge, you may provide both explicitly (like `edge(from, to)`); leave `from` implicit (like `edge(to)`); or leave both implicit. When `from` is implicit, it becomes the coordinate of the last `node`, and if `to` is implicit, the next `node`.

```
#diagram(
  node((0,0), [London]),
  edge("..|>", bend: 20deg),
  node((1,1), [Paris]),
)
```

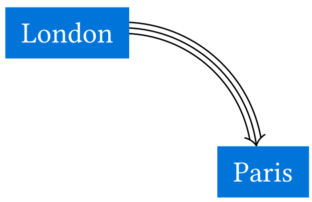


Implicit coordinates can be handy for diagrams in math-mode:

```
#diagram($ L edge("->", bend: #30deg) & P $)
```

However, don't forget you can also use variables in code-mode, which is a more explicit and flexible way to reduce repetition of coordinates.

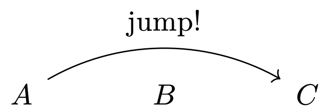
```
#diagram(node-fill: blue, {
  let (dep, arv) = ((0,0), (1,1))
  node(dep, text(white)[London])
  node(arv, text(white)[Paris])
  edge(dep, arv, "==>", bend: 40deg)
})
```



Relative coordinates

You may specify an edge's direction instead of its end coordinate. This can be done with `edge((x, y), (rel: (Δx, Δy)))`, or with string of *directions* for short, e.g., `"u"` for up or `"br"` for bottom right. Any combination of **t**op/**u**p/**n**orth, **b**ottomp/**d**own/**s**outh, **l**eft/**w**est, and **r**ight/**e**ast are allowed. Together with implicit coordinates, this allows you to do things like:

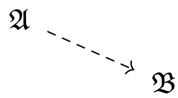
```
#diagram($ A edge("rr", ->, #[jump!], bend: #30deg) & B & C $)
```



Named or labelled coordinates

Another way coordinates can be expressed is through node names. Nodes can be given a `name`, which is a label (not a string) identifying that node. A label as an edge vertex is interpreted as the position of the node with that label.

```
#diagram(
  node((0,0), $frak(A)$, name: <A>),
  node((1,0.5), $frak(B)$, name: <B>),
  edge(<A>, <B>, "-->")
)
```

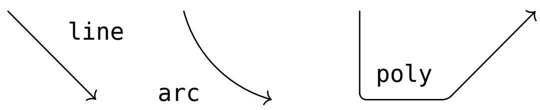


Node names are labels (instead of strings like CeTZ) so that positional arguments to `edge()` are possible to disambiguate by their type. (Node labels are not inserted into the final output, so they do not interfere with other labels in the document.)

Edge types

There are three types of edges: `"line"`, `"arc"`, and `"poly"`. All edges have at least two `vertices`, but `"poly"` edges can have more. If unspecified, `kind` is chosen based on `bend` and the number of `vertices`.

```
#diagram(
  edge((0,0), (1,1), "->", `line`),
  edge((2,0), (3,1), "->", bend: -30deg, `arc`),
  edge((4,0), (4,1), (5,1), (6,0), "->", `poly`),
)
```



All vertices except the first can be relative coordinates (see above), so that in the example above, the `"poly"` edge could also be written in these equivalent ways:

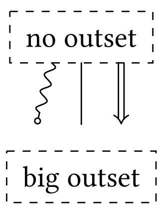
```
edge((4,0), (rel: (0,1)), (rel: (1,0)), (rel: (1,-1)), "->", `poly`)
edge((4,0), "d", "r", "ur", "->", `poly`) // using relative coordinate names
edge((4,0), "d,r,ur", "->", `poly`) // shorthand
```

Only the first and last `vertices` of an edge automatically snap to nodes.

Tweaking where edges connect

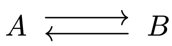
A node's `outset` controls how *close* edges connect to the node's boundary. To adjust *where* along the boundary the edge connects, you can adjust the edge's end coordinates by a fractional amount.

```
#diagram(
  node-stroke: (thickness: .5pt, dash: "dashed"),
  node((0,0), [no outset], outset: 0pt),
  node((0,1), [big outset], outset: 10pt),
  edge((0,0), (0,1)),
  edge((-0.1,0), (-0.4,1), "-o", "wave"), // shifted with fractional coordinates
  edge((0,0), (0,1), "=>", shift: 15pt),  // shifted by a length
)
```



Alternatively, the `shift` option of `edge()` lets you shift edges sideways by an absolute length:

```
#diagram($A edge(->, shift: #3pt) edge(<-, shift: #(-3pt)) & B$)
```



By default, edges which are incident at an angle are automatically adjusted slightly, especially if the node is wide or tall. Aesthetically, things can look more comfortable if edges don't all connect to the node's exact center, but instead spread out a bit. Notice the (subtle) difference the figures below.

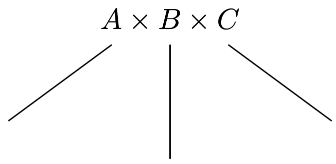


Figure 1: With focus (default)

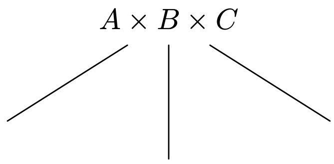
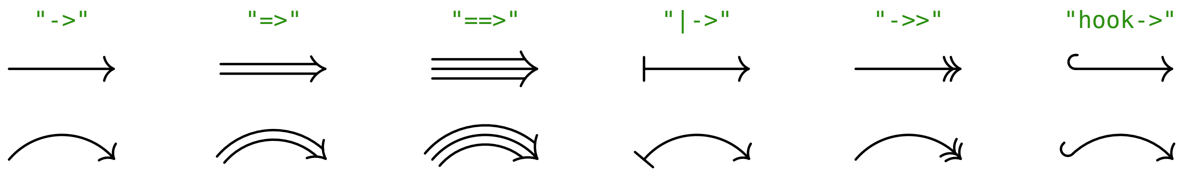


Figure 2: Without defocus

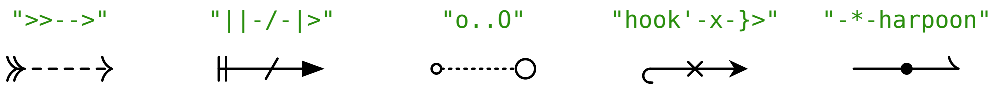
The strength of this adjustment is controlled by the `defocus` option of `node()` (or the `node-defocus` option of `diagram()`).

Marks and arrows

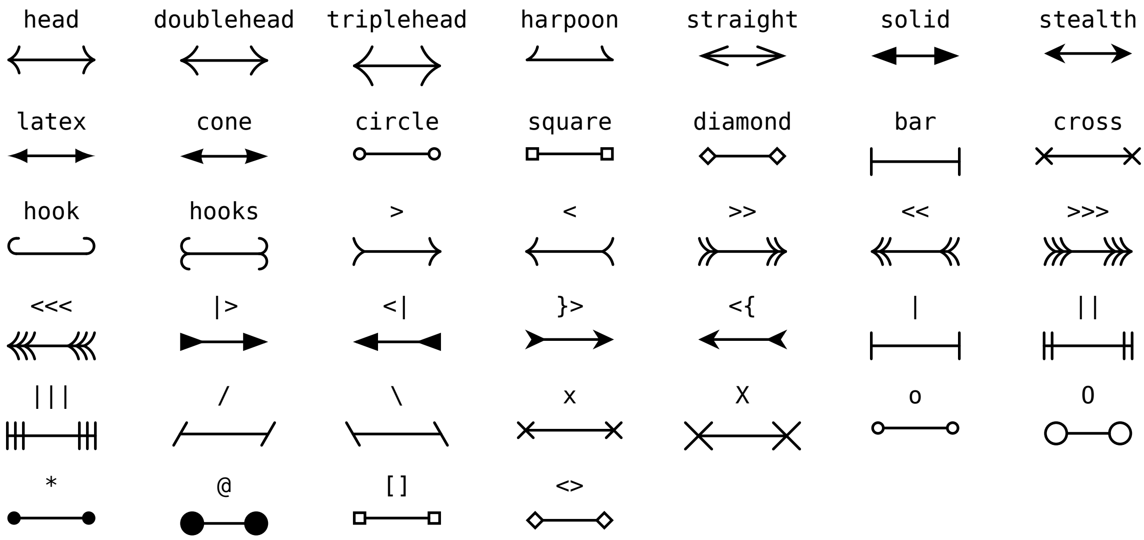
Arrow marks can be specified like `edge(a, b, "-->")` or with the `marks` option of `edge()`. Some mathematical arrow heads are supported, which match $\to$, $\Rightarrow$, $\Rrightarrow$, $\mapsto$, $\twoheadrightarrow$, and $\hookrightarrow$ in the default font.



A few other marks are provided, and all marks can be placed anywhere along the edge.



All the built-in marks are defined in the state variable `fletcher.MARKS`, which you may access with `context fletcher.MARKS.get()`.

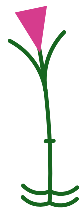


Because it is a state variable, you can modify `fletcher.MARKS` to add or modify mark styles.

Custom marks

While shorthands like `"|=>"` exist for specifying marks and stroke styles, finer control is possible. Marks can be specified by passing an array of *mark objects* to the `marks` option of `edge()`. For example:

```
#diagram(
  edge-stroke: 1.5pt,
  spacing: 25mm,
  edge((0,1), (-0.1,0), bend: -8deg, marks: (
    (inherit: ">>", size: 6, delta: 70deg, sharpness: 65deg),
    (inherit: "head", rev: true, pos: 0.8, sharpness: 0deg, size: 17),
    (inherit: "bar", size: 1, pos: 0.3),
    (inherit: "solid", size: 12, rev: true, stealth: 0.1, fill: red.mix(purple)),
  ), stroke: green.darken(50%)),
)
```



In fact, shorthands like `"|=>"` are expanded with `interpret-marks-arg()` into a form more like the example above. More precisely, `edge(from, to, "|=>")` is equivalent to:

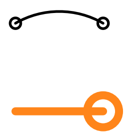
```
context edge(from, to, ..fletcher.interpret-marks-arg("|=>"))
```

If you want to explore the internals of mark objects, you might find it handy to inspect the output of `context fletcher.interpret-marks-arg(..)` with various mark shorthands as input.

Mark objects

A *mark object* is a dictionary with, at the very least, a `draw` entry containing the CeTZ objects to be drawn on the edge. These CeTZ objects are translated and scaled to fit the edge; the mark's center should be at the origin, and the stroke's thickness is defined as the unit length. For example, here is a basic circle mark:

```
#import cetz.draw
#let my-mark = (
  draw: draw.circle((0,0), radius: 2, fill: none)
)
#diagram(
  edge((0,0), (1,0), stroke: 1pt, marks: (my-mark, my-mark), bend: 30deg),
  edge((0,1), (1,1), stroke: 3pt + orange, marks: (none, my-mark)),
)
```

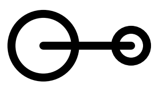


A mark object can contain arbitrary parameters, which may depend on parameters defined earlier by being written as a *function* of the mark object. For example, the mark above could also be written as:

```
#let my-mark = (
  size: 2,
  draw: mark => draw.circle((0,0), radius: mark.size, fill: none)
)
```

This form makes it easier to change the size without modifying the `draw` function, for example:

```
#diagram(edge(stroke: 3pt, marks: (my-mark + (size: 4), my-mark)))
```



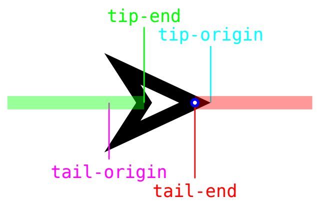
Internally, marks are passed to `resolve-mark()`, which ensures all entries are evaluated to final values.

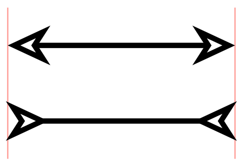
Special mark properties

A mark object may contain any properties, but some have special functions.

Name	*Description*	*Default*
`inherit`	The name of a mark in `fletcher.MARKS` to inherit properties from. This can be used to make mark aliases, for instance, `"<"` is defined as `(inherit: "head", rev: true)`.	
`draw`	As described above, this contains the final CeTZ objects to be drawn. Objects should be centered at $(0, 0)$ and be scaled so that one unit is the stroke thickness. The default `stroke` and `fill` is inherited from the edge's style.	
`pos`	Location of the mark along the edge, from `0` (start) to `1` (end).	`auto`
`fill` `stroke`	The default fill and stroke styles for CeTZ objects returned by `draw`. If `none`, polygons will not be filled/stroked by default, and if `auto`, the style is inherited from the edge's stroke style.	`auto`
`rev`	Whether to reverse the mark so it points backwards.	`false`
`flip`	Whether to reflect the mark across the edge; the difference between ⊂—— and ⊂——, for example. A suffix `'` in the name, such as `"hook'"`, results in a flip.	`false`
`scale`	Overall scaling factor. See also the `mark-scale` option of `edge()`.	`100%`
`extrude`	Whether to duplicate the mark and draw it offset at each extrude position. For example, `(inherit: "head", extrude: (-5, 0, 5))` looks like ⟶⋙.	`(0,)`
`tip-origin` `tail-origin`	These two properties control the x coordinate of the point of the mark, relative to $(0, 0)$. If the mark is acting as a tip (⟶ or ⟵) then `tip-origin` applies, and `tail-origin` applies when the mark is a tail (—< or >—). See `mark-debug()`.	`0`
`tip-end` `tail-end`	These control the x coordinate at which the edge's stroke terminates, relative to $(0, 0)$. See `mark-debug()`.	`0`
`cap-offset`	A function `(mark, y) => x` returning the x coordinate at which the edge's stroke terminates relative to `tip-end` or `tail-end`, as a function of the y coordinate. This is relevant for `extrude`d edges. See `cap-offset()`.	

The last few properties control the fine behaviours of how marks connect to the target point and to the edge's stroke. Briefly, a mark has four possibly-distinct center points. It is easier to show than to tell:



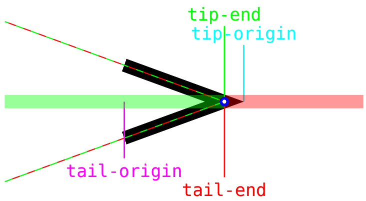


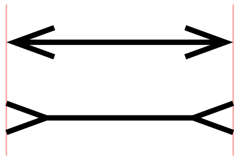
See `mark-debug()` and `cap-offset()` for details.

Detailed example

As a complete example, here is the implementation of a straight arrowhead in `src/default-marks.typ`:

```
#import cetz.draw
#let straight = (
  size: 8,
  sharpness: 20deg,
  tip-origin: mark => 0.5/calc.sin(mark.sharpness),
  tail-origin: mark => -mark.size*calc.cos(mark.sharpness),
  fill: none,
  draw: mark => {
    draw.line(
      (180deg + mark.sharpness, mark.size), // polar cetz coordinate
      (0, 0),
      (180deg - mark.sharpness, mark.size),
    )
  },
  cap-offset: (mark, y) => calc.tan(mark.sharpness + 90deg)*calc.abs(y),
)

#set align(center)
#fletcher.mark-debug(straight)
#fletcher.mark-demo(straight)
```





Custom mark shorthands

While you can pass custom mark objects directly to the `marks` option of `edge()`, this can get annoying if you use the same mark often. In these cases, you can define your own mark shorthands.

Mark shorthands such as `"hook->"` search the state variable `fletcher.MARKS` for defined mark names.

```
#context fletcher.MARKS.get().at(">")
```

`(inherit: "head", rev: false)`

With a bit of care, you can modify the `MARKS` state like so:

```
// this is what the default marks look like
#diagram(spacing: 3cm, edge("<->", stroke: 1.5pt))

#fletcher.MARKS.update(m => m + (
  "<": (inherit: "stealth", rev: true),
  ">": (inherit: "stealth", rev: false),
  "multi": (
    inherit: "straight",
    draw: mark => fletcher.cetz.draw.line(
      (0, +mark.size*calc.sin(mark.sharpness)),
      (-mark.size*calc.cos(mark.sharpness), 0),
      (0, -mark.size*calc.sin(mark.sharpness)),
    ),
  )
))

// subsequent diagrams will use your updated marks
#diagram(spacing: 3cm, edge("multi->-multi", stroke: 1.5pt + eastern))
```

Here, we redefined which mark style the `"<"` and `">"` shorthands refer to, and added an entirely new mark style with the shorthand `"multi"`.

Finally, I will restore the default state so as not to affect the rest of this manual:

```
#fletcher.MARKS.update(fletcher.DEFAULT_MARKS) // restore to built-in mark styles
```

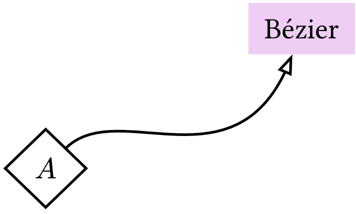
CeTZ integration

Fletcher's drawing capabilities are deliberately restricted to a few simple building blocks. However, an escape hatch is provided with the `render` option of `diagram()` so you can intercept diagram data and draw things using CeTZ directly.

Bézier edges

Here is an example of how you might hack together a Bézier edge using the same functions that `fletcher` uses internally to anchor edges to nodes:

```
#diagram(
  node((0,1), $A$, stroke: 1pt, shape: fletcher.shapes.diamond),
  node((2,0), [Bézier], fill: purple.lighten(80%)),

  render: (grid, nodes, edges, options) => {
    // cetz is also exported as fletcher.cetz
    cetz.canvas({
      // this is the default code to render the diagram
      fletcher.draw-diagram(grid, nodes, edges, debug: options.debug)

      // retrieve node data by coordinates
      let n1 = fletcher.find-node-at(nodes, (0,1))
      let n2 = fletcher.find-node-at(nodes, (2,0))

      let out-angle = 45deg
      let in-angle = -110deg

      fletcher.get-node-anchor(n1, out-angle, p1 => {
        fletcher.get-node-anchor(n2, in-angle, p2 => {
          // make some control points
          let c1 = (to: p1, rel: (out-angle, 10mm))
          let c2 = (to: p2, rel: (in-angle, 20mm))
          cetz.draw.bezier(
            p1, p2, c1, c2,
            mark: (end: ">") // cetz-style mark
          )
        })
      })

    })
  }
)
```



Touying integration

You can create incrementally-revealed diagrams in Touying presentation slides by defining the following `touying-reducer`:

```
#import "@preview/touying:0.2.1": *
#let diagram = touying-reducer.with(reduce: fletcher.diagram, cover: fletcher.hide)
#let (init, slide) = utils.methods(s)
#show: init

#slide[
  Slide with animated figure:
  #diagram(
    node-stroke: .1em,
    node-fill: gradient.radial(blue.lighten(80%), blue,
      center: (30%, 20%), radius: 80%),
    spacing: 4em,
    edge((-1,0), "r", "-|>", `open(path)`, label-pos: 0, label-side: center),
    node((0,0), `reading`, radius: 2em),
    pause,
    edge((0,0), (0,0), `read()`, "--|>", bend: 130deg),
    edge(`read()`, "-|>"),
    node((1,0), `eof`, radius: 2em),
    pause,
    edge(`close()`, "-|>"),
    node((2,0), `closed`, radius: 2em, extrude: (-2.5, 0)),
    edge((0,0), (2,0), `close()`, "-|>", bend: -40deg),
  )
]
```

Reference

Main functions

`diagram()`

Draw a diagram containing `node()`s and `edge()`s.

```
diagram(
  ..args: array,
  debug: bool 1 2 3,
  axes: pair of directions,
  spacing: length pair of lengths,
  cell-size: length pair of lengths,
  edge-stroke: stroke,
  node-stroke: stroke none,
  edge-corner-radius: length none,
  node-corner-radius: length none,
  node-inset: length pair of lengths,
  node-outset: length pair of lengths,
  node-fill: paint,
  node-defocus: number,
  label-sep: length,
  mark-scale: percent,
  crossing-fill: paint,
  crossing-thickness: number,
  render: function,
)
```

`..args` `array`

Content to draw in the diagram, including nodes and edges.

The results of `node()` and `edge()` can be *joined*, meaning you can specify them as separate arguments, or in a block:

```
#diagram(
  // one object per argument
  node((0, 0), $A$),
  node((1, 0), $B$),
  {
    // multiple objects in a block
    // can use scripting, loops, etc
    node((2, 0), $C$)
    node((3, 0), $D$)
  },
  for x in range(4) { node((x, 1) [#x]) },
)
```

Nodes and edges can also be specified in math-mode.

```
#diagram($
  A & B \          // two nodes at (0,0) and (1,0)
  C edge(->) & D \ // an edge from (0,1) to (1,1)
  node(sqrt(pi), stroke: #1pt) // a node with options
$)
```

debug `bool` or `1` or `2` or `3`

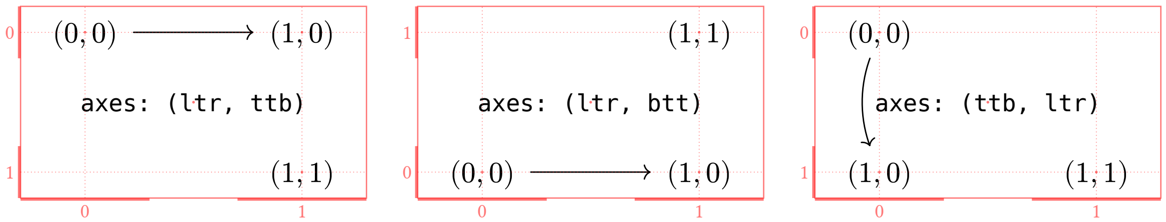
Level of detail for drawing debug information. Level `1` or `true` shows a coordinate grid; higher levels show bounding boxes and anchors, etc.

Default: `false`

axes `pair of directions`

The orientation of the diagram's axes.

This defines the elastic coordinate system used by nodes and edges. To make the y coordinate increase up the page, use `(ltr, btt)`. For the matrix convention `(row, column)`, use `(ttb, ltr)`.



Default: `(ltr, ttb)`

spacing `length` or `pair of lengths`

Gaps between rows and columns. Ensures that nodes at adjacent grid points are at least this far apart (measured as the space between their bounding boxes).

Separate horizontal/vertical gutters can be specified with `(x, y)`. A single length `d` is short for `(d, d)`.

Default: `3em`

cell-size `length` or `pair of lengths`

Minimum size of all rows and columns. A single length `d` is short for `(d, d)`.

Default: `0pt`

edge-stroke `stroke`

Default value of the `stroke` option of `edge()`. By default, this is chosen to match the thickness of mathematical arrows such as $A \to B$ in the current font size.

The default stroke is folded with the stroke specified for the edge. For example, if `edge-stroke` is `1pt` and the `stroke` option of `edge()` is `red`, then the resulting stroke is `1pt + red`.

Default: `0.048em`

`node-stroke` `stroke` or `none`

Default value of the `stroke` option of `node()`.

The default stroke is folded with the stroke specified for the node. For example, if `node-stroke` is `1pt` and the `stroke` option of `node()` is `red`, then the resulting stroke is `1pt + red`.

Default: `none`

`edge-corner-radius` `length` or `none`

Default value of the `corner-radius` option of `edge()`.

Default: `2.5pt`

`node-corner-radius` `length` or `none`

Default value of the `corner-radius` option of `node()`.

Default: `none`

`node-inset` `length` or `pair of lengths`

Default value of the `inset` option of `node()`.

Default: `6pt`

`node-outset` `length` or `pair of lengths`

Default value of the `outset` option of `node()`.

Default: `0pt`

`node-fill` `paint`

Default value of the `fill` option of `node()`.

Default: `none`

`node-defocus` `number`

Default value of the `defocus` option of `node()`.

Default: `0.2`

`label-sep` `length`

Default value of the `label-sep` option of `edge()`.

Default: `0.2em`

mark-scale `percent`

Default value of the `mark-scale` option of `edge()`.

Default: `100%`

crossing-fill `paint`

Color to use behind connectors or labels to give the illusion of crossing over other objects. See the `crossing-fill` option of `edge()`.

Default: `white`

crossing-thickness `number`

Default thickness of the occlusion made by crossing connectors. See `crossing-thickness`.

Default: `5`

render `function`

After the node sizes and grid layout have been determined, the `render` function is called with the following arguments:

- `grid`: a dictionary of the row and column widths and positions;
- `nodes`: an array of nodes (dictionaries) with computed attributes (including size and physical coordinates);
- `edges`: an array of connectors (dictionaries) in the diagram; and
- `options`: other diagram attributes.

This callback is exposed so you can access the above data and draw things directly with CeTZ.

Default:
```
(grid, nodes, edges, options) => {
cetz.canvas(draw-diagram(grid, nodes, edges, debug: options.debug))
}
```

node()

Draw a labelled node in a diagram which can connect to edges.

```
node(
  ..args,
  pos: coordinate,
  name: label none,
  label: content,
  inset: length auto,
  outset: length auto,
  fill: paint,
  stroke: stroke,
  extrude: array,
  width: length auto,
  height: length auto,
  radius,
  enclose: array,
  corner-radius: length,
  shape: rect circle function auto,
  defocus: number,
  layer: number auto,
  post: function,
)
```

pos `coordinate`

Dimensionless "elastic coordinates" `(x, y)` of the node.

See the options of `diagram()` to control the physical scale of elastic coordinates.

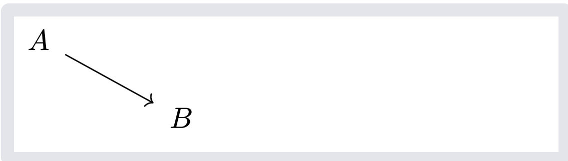
Default: `auto`

name `label` or `none`

An optional name to give the node.

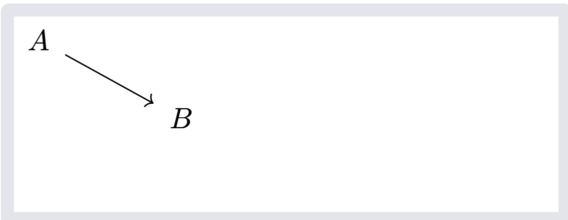
Names can sometimes be used in place of coordinates. For example:

```
fletcher.diagram(
  node((0,0), $A$, name: <A>),
  node((1,0.6), $B$, name: <B>),
  edge(<A>, <B>, "->"),
)
```



Note that you can also just use variables to refer to coordinates:

```
fletcher.diagram({
  let A = (0,0)
  let B = (1,0.6)
  node(A, $A$)
  node(B, $B$)
  edge(A, B, "->")
})
```



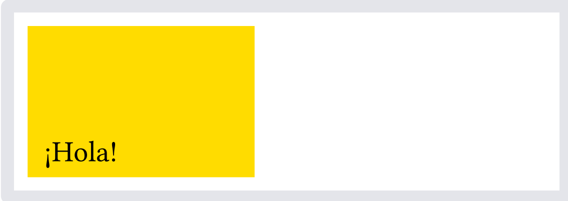
Default: `none`

`label` `content`

Content to display inside the node.

If a node is larger than its label, you can wrap the label in `align()` to control the label alignment within the node.

```
diagram(
  node((0,0), align(bottom + left)[¡Hola!],
    width: 3cm, height: 2cm, fill: yellow),
)
```



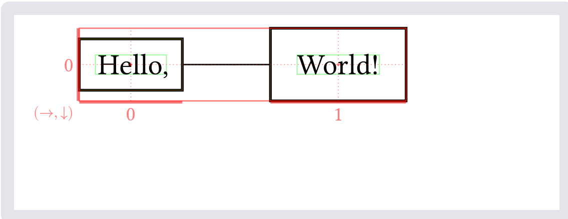
Default: `none`

`inset` `length` or `auto`

Padding between the node's content and its outline.

In debug mode, the inset is visualised by a thin green outline.

```
diagram(
  debug: 3,
  node-stroke: 1pt,
  node((0,0), [Hello,]),
  edge(),
  node((1,0), [World!], inset: 10pt),
)
```



Default: `auto`

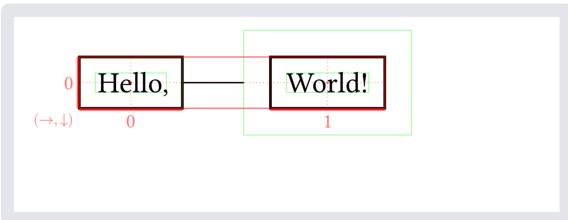
`outset` `length` or `auto`

Margin between the node's bounds to the anchor points for connecting edges.

This does not affect node layout, only how closely edges connect to the node.

In debug mode, the outset is visualised by a thin green outline.

```
diagram(
  debug: 3,
  node-stroke: 1pt,
  node((0,0), [Hello,]),
  edge(),
  node((1,0), [World!], outset: 10pt),
)
```



Default: `auto`

fill `paint`

Fill style of the node. The fill is drawn within the node outline as defined by the first `extrude` value.

Defaults to the `node-fill` option of `diagram()`.

Default: `auto`

stroke `stroke`

Stroke style for the node outline.

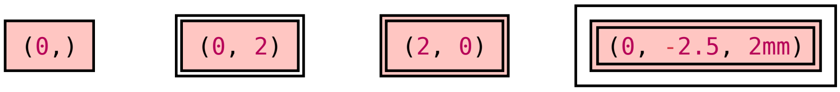
Defaults to the `node-stroke` option of `diagram()`.

Default: `auto`

extrude `array`

Draw strokes around the node at the given offsets to obtain a multi-stroke effect. Offsets may be numbers (specifying multiples of the stroke's thickness) or lengths.

The node's fill is drawn within the boundary defined by the first offset in the array.



See also the `extrude` option of `edge()`.

Default: `(0,)`

width `length` or `auto`

Width of the node. If `auto`, the node's width is the width of the node `label`, plus twice the `inset`.

If the width is not `auto`, you can use `align` to control the placement of the node's `label`.

Default: `auto`

height `length` or `auto`

Height of the node. If `auto`, the node's height is the height of the node `label`, plus twice the `inset`.

If the height is not `auto`, you can use `align` to control the placement of the node's `label`.

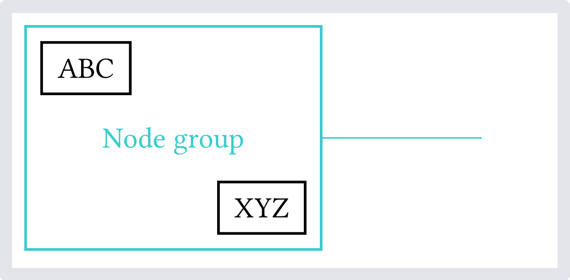
Default: `auto`

enclose `array`

Positions or names of other nodes to enclose by enlarging this node.

If given, causes the node to resize so that its bounding rectangle surrounds the given nodes. The center `pos` does not affect the node's position if `enclose` is given, but still affects connecting edges.

```
diagram(
  node-stroke: 1pt,
  node((0,0), [ABC], name: <A>),
  node((1,1), [XYZ], name: <Z>),
  node(
    text(teal)[Node group], stroke: teal,
    enclose: (<A>, <Z>), name: <group>),
  edge(<group>, (3,0.5), stroke: teal),
)
```



Default: `()`

`corner-radius` `length`

Radius of rounded corners, if supported by the node `shape`.

Defaults to the `node-corner-radius` option of `diagram()`.

Default: `auto`

`shape` `rect` or `circle` or `function` or `auto`

Shape to draw for the node. If `auto`, one of `rect` or `circle` is chosen depending on the aspect ratio of the node's label.

Other shapes are defined in the `fletcher.shapes` submodule, including `cetz`, `draw`, `vector`, `rect`, `circle`, `ellipse`, `pill`, `parallelogram`, `diamond`, `triangle`, `house`, `chevron`, `hexagon`, and `octagon`.

Custom shapes should be specified as a function `(node, extrude, ..parameters) => (..)` which returns `cetz` objects.

- The `node` argument is a dictionary containing the node's attributes, including its dimensions (`node.size`), and other options (such as `node.corner-radius`).
- The `extrude` argument is a length which the shape outline should be extruded outwards by. This serves two functions: to support automatic edge anchoring with a non-zero node `outset`, and to create multi-stroke effects using the `extrude` node option.

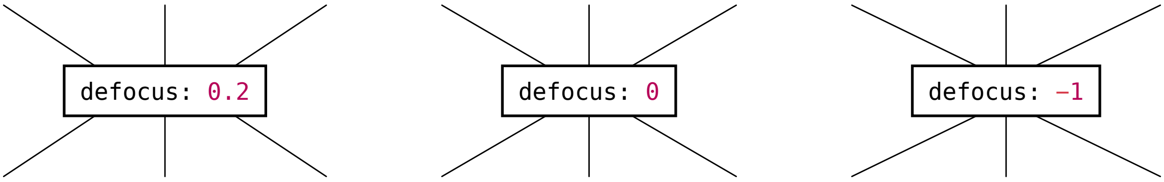
See the `src/shapes.typ` source file for examples.

Default: `auto`

`defocus` `number`

Strength of the "defocus" adjustment for connectors incident with this node.

This affects how connectors attach to non-square nodes. If `0`, the adjustment is disabled and connectors are always directed at the node's exact center.



Defaults to the `node-defocus` option of `diagram()`.

Default: `auto`

`layer` `number` or `auto`

Layer on which to draw the node.

Objects on a higher `layer` are drawn on top of objects on a lower `layer`. Objects on the same layer are drawn in the order they are passed to `diagram()`.

By default, nodes are drawn on layer `0` unless they `enclose` points, in which case `layer` defaults to `-1`.

Default: `auto`

`post` `function`

Callback function to intercept `cetz` objects before they are drawn to the canvas.

This can be used to hide elements without affecting layout (for use with Touying, for example). The `hide()` function also helps for this purpose.

Default: `x => x`

edge()

Draw a connecting line or arc in an arrow diagram.

```
edge(
  ..args: any ,
  vertices: array ,
  label: content ,
  label-side: left  right  center ,
  label-pos: number ,
  label-sep: length ,
  label-anchor: anchor ,
  label-fill: bool  paint ,
  stroke: stroke ,
  dash: string ,
  decorations: none  string  function ,
  extrude: array ,
  shift: length  number  pair ,
  kind: string ,
  bend: angle ,
  corner: none  left  right ,
  corner-radius: length  none ,
  marks: array ,
  mark-scale: percent ,
  crossing: bool ,
  crossing-thickness: number ,
  crossing-fill: paint ,
  snap-to: pair ,
  layer: number ,
  post: function ,
)
```

`..args` any

An edge's positional arguments may specify:

- the edge's `vertices`
- the `label` content
- `marks` and other style options

Vertex coordinates must come first, and are optional:

```
edge(from, to, ..) // explicit start and end nodes
edge(to, ..) // start node chosen automatically based on last node specified
edge(..) // both nodes chosen automatically depending on adjacent nodes
edge(from, v1, v2, ..vs, to, ..) // a multi-segmented edge
```

All coordinates except the start point can be relative (a dictionary of the form `(rel: (Δx, Δy))` or a string containing the characters {`l`, `r`, `u`, `d`, `t`, `b`, `n`, `e`, `s`, `w`}).

An edge's `marks` and `label` can be also specified as positional arguments. They are disambiguated by guessing based on the types. For example, the following are equivalent:

```
edge((0,0), (1,0), $f$, "->")
edge((0,0), (1,0), "->", $f$)
edge((0,0), (1,0), $f$, marks: "->")
edge((0,0), (1,0), "->", label: $f$)
edge((0,0), (1,0), label: $f$, marks: "->")
```

Additionally, some common options are given flags that may be given as string positional arguments. These are `"dashed"`, `"dotted"`, `"double"`, `"triple"`, `"crossing"`, `"wave"`, `"zigzag"`, and `"coil"`. For example, the following are equivalent:

```
edge((0,0), (1,0), $f$, "wave", "crossing")
edge((0,0), (1,0), $f$, decorations: "wave", crossing: true)
```

`vertices` `array`

Array of (at least two) coordinates for the edge.

Vertices can also be specified as leading positional arguments, but if so, the `vertices` option must be empty. If the number of vertices is greater than two, `kind` defaults to `"poly"`.

Default: `()`

`label` `content`

Content for the edge label. See the `label-pos` and `label-side` options to control the position (and `label-sep` and `label-anchor` for finer control).

Default: `none`

`label-side` `left` or `right` or `center`

Which side of the edge to place the label on, viewed as you walk along it from base to tip.

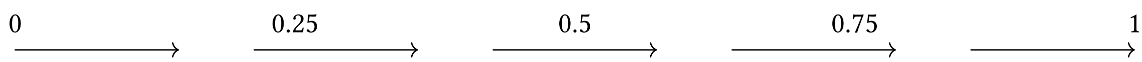
If `center`, then the label is placed directly on the edge and `label-fill` defaults to `true`. When `auto`, a value of `left` or `right` is automatically chosen so that the label is:

- roughly above the connector, in the case of straight lines; or
- on the outside of the curve, in the case of arcs.

Default: `auto`

`label-pos` `number`

Position of the label along the connector, from the start to end (from `0` to `1`).

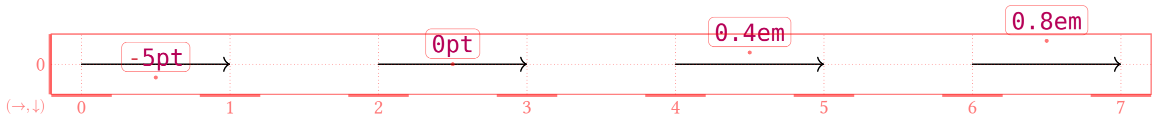


Default: `0.5`

`label-sep` `length`

Separation between the connector and the label anchor.

With the default anchor (automatically set to `"bottom"` in this case):



With `label-anchor` set to `"center"`:

Set `debug` to `2` or higher to see label anchors and outlines as seen here.

Default: `auto`

`label-anchor` `anchor`

The anchor point to place the label at, such as `"top-right"`, `"center"`, `"bottom"`, etc. If `auto`, the anchor is automatically chosen based on `label-side` and the angle of the connector.

Default: `auto`

`label-fill` `bool` or `paint`

The background fill for the label. If `true`, defaults to the value of `crossing-fill`. If `false` or `none`, no fill is used. If `auto`, then defaults to `true` if the label is covering the edge (`label-side: center`).

Default: `auto`

`stroke` `stroke`

Stroke style of the edge. Arrows/marks scale with the stroke thickness (and with `mark-scale`).

Default: `auto`

`dash` `string`

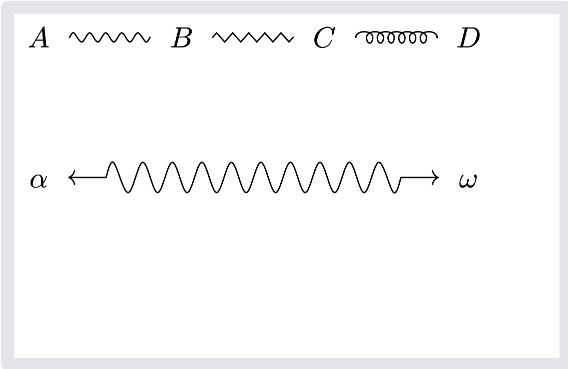
The stroke's dash style. This is also set by some mark styles. For example, setting `marks: "<..>"` applies `dash: "dotted"`.

Default: `none`

decorations `none` or `string` or `function`

Apply a CeTZ path decoration to the stroke. Preset options are `"wave"`, `"zigzag"`, and `"coil"` (which may also be passed as convenience positional arguments), but a decoration function may also be specified.

```
diagram(
  $
    A edge("wave") &
    B edge("zigzag") &
    C edge("coil") & D \
    alpha &&& omega
  $,
  edge((0,1), (3,1), "<->", decorations:
    cetz.decorations.wave
      .with(amplitude: .4)
  )
)
```

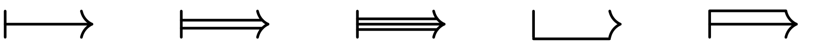


Default: `none`

extrude `array`

Draw a separate stroke for each extrusion offset to obtain a multi-stroke effect. Offsets may be numbers (specifying multiples of the stroke's thickness) or lengths.

`(0,)` `(-1.5, 1.5)` `(-2, 0, 2)` `(-0.5em,)` `(0, 5pt)`



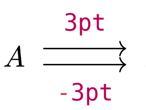
Notice how the ends of the line need to shift a little depending on the mark. This offset is computed with `cap-offset()`.

See also the `extrude` option of `node()`.

Default: `(0,)`

shift `length` or `number` or `pair`

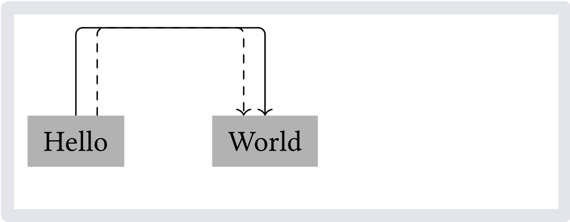
Amount to shift the edge sideways by, perpendicular to its direction. A pair `(from, to)` controls the shifts at each end of the edge independently, and a single shift `s` is short for `(s, s)`. Shifts can absolute lengths (e.g., `5pt`) or coordinate differences (e.g., `0.1`).



If an edge has many vertices, the shifts only affect the first and last segments of the edge.

```
diagram(
  node-fill: luma(70%),
  node((0,0), [Hello]),
  edge("u,r,d", "->"),
  edge("u,r,d", "-->", shift: 8pt),
```

```
  node((1,0), [World]),
)
```



Default: `0pt`

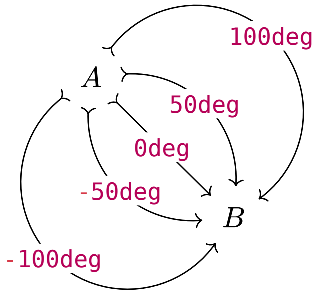
`kind` `string`

The kind of the edge, one of `"line"`, `"arc"`, or `"poly"`. This is chosen automatically based on the presence of other options (`bend` implies `"arc"`, `corner` or additional vertices implies `"poly"`).

Default: `auto`

`bend` `angle`

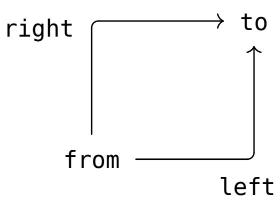
Edge curvature. If `0deg`, the connector is a straight line; positive angles bend clockwise.



Default: `0deg`

`corner` `none` or `left` or `right`

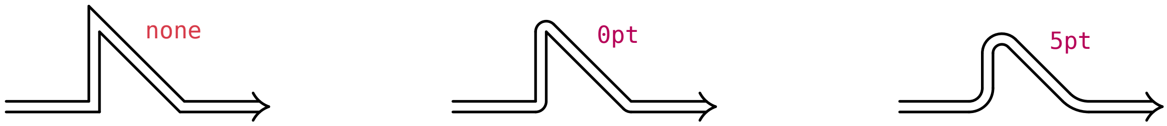
Whether to create a right-angled corner, turning `left` or `right`. (Bending right means the corner sticks out to the left, and vice versa.)



Default: `none`

`corner-radius` `length` or `none`

Radius of rounded corners for edges with multiple segments. Note that `none` is distinct from `0pt`.



This length specifies the corner radius for right-angled bends. The actual radius is smaller for acute angles and larger for obtuse angles to balance things visually. (Trust me, it looks naff otherwise!)

If `auto`, defaults to the `edge-corner-radius` option of `diagram()`.

Default: `auto`

`marks` `array`

The marks (arrowheads) to draw along an edge's stroke. This may be:

- A shorthand string such as `"->"` or `"hook'-/->>"`. Specifically, shorthand strings are of the form $M_1 L M_2$ or $M_1 L M_2 L M_3$, etc, where

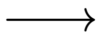
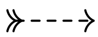
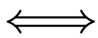
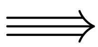
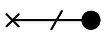
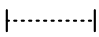
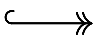
$$M_i \in \texttt{fletcher.MARKS} = \left\{\begin{array}{ccccc} \texttt{head}, & \texttt{doublehead}, & \texttt{triplehead}, & \texttt{harpoon}, & \texttt{straight}, \\ \texttt{solid}, & \texttt{stealth}, & \texttt{latex}, & \texttt{cone}, & \texttt{circle}, \\ \texttt{square}, & \texttt{diamond}, & \texttt{bar}, & \texttt{cross}, & \texttt{hook}, \\ \texttt{hooks}, & \texttt{>}, & \texttt{<}, & \texttt{>>}, & \texttt{<<}, \\ \texttt{>>>}, & \texttt{<<<}, & \texttt{|>}, & \texttt{<|}, & \texttt{\}>}, \\ \texttt{<\{}, & \texttt{|}, & \texttt{||}, & \texttt{|||}, & \texttt{/}, \\ \texttt{\textbackslash}, & \texttt{x}, & \texttt{X}, & \texttt{o}, & \texttt{O}, \\ \texttt{*}, & \texttt{@}, & \texttt{[]}, & \texttt{<>}, & \end{array}\right\}$$

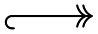
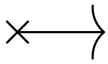
  is a mark name and

$$L \in \texttt{fletcher.LINE\_ALIASES} = \{\texttt{-}, \texttt{=}, \texttt{==}, \texttt{--}, \texttt{..}, \texttt{\textasciitilde}\}$$

  is the line style.
- An array of marks, where each mark is specified by name of as a *mark object* (dictionary of parameters with a `draw` entry).

Shorthands are expanded into other arguments. For example, `edge(p1, p2, "=>")` is short for `edge(p1, p2, marks: (none, "head"), "double")`, or more precisely, the result of `edge(p1, p2, ..fletcher.interpret-marks-arg("=>"))`.

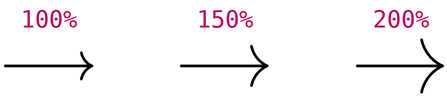
<table>
<tr><th>Result</th><th>Value of <code>marks</code></th></tr>
<tr><td></td><td><code>"->"</code></td></tr>
<tr><td></td><td><code>">>-->"</code></td></tr>
<tr><td></td><td><code>"<=>"</code></td></tr>
<tr><td></td><td><code>"==>"</code></td></tr>
<tr><td></td><td><code>"->>-"</code></td></tr>
<tr><td></td><td><code>"x-/-@"</code></td></tr>
<tr><td></td><td><code>"|..|"</code></td></tr>
<tr><td></td><td><code>"hook->>"</code></td></tr>
</table>

	`"hook'->>"`
	`"||-*-harpoon'"`
	`("X", (inherit: "head", size: 15, sharpness: 40deg))`
	`((inherit: "circle", pos: 0.5, fill: auto),)`

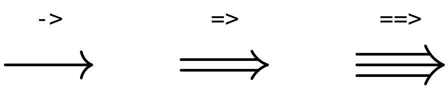
Default: `()`

`mark-scale` `percent`

Scale factor for marks or arrowheads, relative to the `stroke` thickness. See also the `mark-scale` option of `diagram()`.



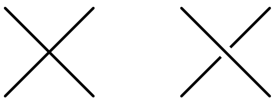
Note that the default arrowheads scale automatically with double and triple strokes:



Default: `100%`

`crossing` `bool`

If `true`, draws a backdrop of color `crossing-fill` to give the illusion of lines crossing each other.

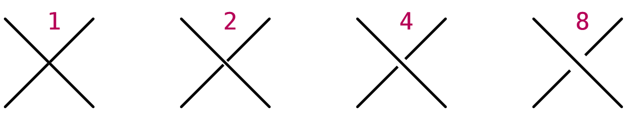


You can also pass `"crossing"` as a positional argument as a shorthand for `crossing: true`.

Default: `false`

`crossing-thickness` `number`

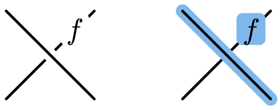
Thickness of the “crossing” background stroke (applicable if `crossing` is `true`) in multiples of the normal stroke’s thickness. Defaults to the `crossing-thickness` option of `diagram()`.



Default: `auto`

`crossing-fill` `paint`

Color to use behind connectors or labels to give the illusion of crossing over other objects. Defaults to `crossing-fill`.



Default: `auto`

`snap-to` `pair`

The nodes the start and end of an edge should snap to. Each node can be a position or node `name`, or `none` to disable snapping.

By default, an edge's first and last `vertices` snap to nearby nodes. This option can be used in case automatic snapping fails (if there are many nodes close together, for example.)

Default: `(auto, auto)`

`layer` `number`

Layer on which to draw the edge.

Objects on a higher `layer` are drawn on top of objects on a lower `layer`. Objects on the same layer are drawn in the order they are passed to `diagram()`.

Default: `0`

`post` `function`

Callback function to intercept `cetz` objects before they are drawn to the canvas.

This can be used to hide elements without affecting layout (for use with Touying, for example). The `hide()` function also helps for this purpose.

Default: `x => x`

Behind the scenes

`marks.typ`

The default marks are defined in the `fletcher.MARKS` dictionary with keys: `head`, `doublehead`, `triplehead`, `harpoon`, `straight`, `solid`, `stealth`, `latex`, `cone`, `circle`, `square`, `diamond`, `bar`, `cross`, `hook`, `hooks`, `>`, `<`, `>>`, `<<`, `>>>`, `<<<`, `|>`, `<|`, `}>`, `<{`, `|`, `||`, `|||`, `/`, `\`, `x`, `X`, `o`, `O`, `*`, `@`, `[]`, and `<>`.

- `cap-offset()`
- `resolve-mark()`
- `draw-mark()`
- `mark-debug()`

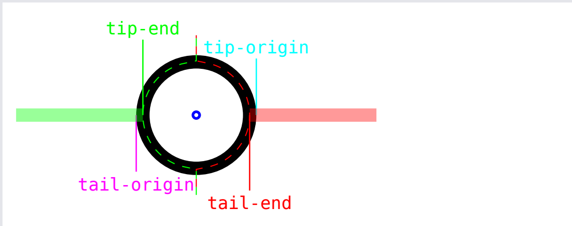
`cap-offset()`

For a given mark, determine where that the stroke should terminate at, relative to the mark's origin point, as a function of the shift.

Imagine the tip-origin of the mark is at $(x, y) = (0, 0)$. A stroke along the line $y =$ shift coming from $x = -\infty$ terminates at $x =$ offset, where offset is the result of this function. Units are in multiples of stroke thickness.

This is used to correctly implement multi-stroke marks, e.g., $\Longleftrightarrow$. The function `mark-debug()` can help visualise a mark's cap offset.

```
fletcher.mark-debug("O")
```



The dashed green line shows the stroke tip end as a function of y, and the dashed red line shows where the stroke ends if the mark is acting as a tail.

```
cap-offset(mark, shift)
```

`resolve-mark()`

Resolve a mark dictionary by applying inheritance, adding any required entries, and evaluating any closure entries.

```
context fletcher.resolve-mark((
  a: 1,
  b: 2,
  c: mark => mark.a + mark.b,
))
```

```
(
  a: 1,
  b: 2,
  c: 3,
  rev: false,
  flip: false,
  scale: 100%,
  extrude: (0,),
  tip-end: 0,
  tail-end: 0,
  tip-origin: 0,
  tail-origin: 0,
)
```

```
resolve-mark(mark, defaults)
```

draw-mark()

Draw a mark at a given position and angle

```
draw-mark(
  mark: dictionary,
  stroke: stroke,
  origin: point,
  angle: angle,
  debug: bool,
)
```

mark `dictionary`

Mark object to draw. Must contain a `draw` entry.

stroke `stroke`

Stroke style for the mark. The stroke's paint is used as the default fill style.

Default: `1pt`

origin `point`

Coordinate of the mark's origin (as defined by `tip-origin` or `tail-origin`).

Default: `(0,0)`

angle `angle`

Angle of the mark, `0deg` being →, counterclockwise.

Default: `0deg`

debug `bool`

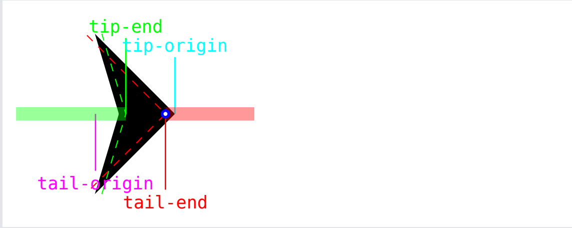
Whether to draw the origin points.

Default: `false`

mark-debug()

Visualise a mark's anatomy.

```
context {
  let mark = fletcher.MARKS.get().stealth
  // make a wide stealth arrow
  mark += (angle: 45deg)
  fletcher.mark-debug(mark)
}
```



- Green/left stroke: the edge's stroke when the mark is at the tip.
- Red/right stroke: edge's stroke if the mark is at the start acting as a tail.
- Blue-white dot: the origin point $(0,0)$ in the mark's coordinate frame.
- `tip-origin`: the x-coordinate of the point of the mark's tip.
- `tail-origin`: the x-coordinate of the mark's tip when it is acting as a reversed tail mark.
- `tip-end`: The x-coordinate of the end point of the edge's stroke (green stroke).
- `tail-end`: The x-coordinate of the end point of the edge's stroke when acting as a tail mark (red stroke).
- Dashed green/red lines: The stroke end points as a function of y. This is controlled by the special `cap-offset` mark property and is used for multi-stroke effects like $\Rrightarrow$. See `cap-offset()`.

This is mainly useful for designing your own marks.

```
mark-debug(
  mark: string dictionary,
  stroke: stroke,
  show-labels: bool,
  show-offsets: bool,
  offset-range: number,
)
```

mark `string` or `dictionary`

The mark name or dictionary.

stroke `stroke`

The stroke style, whose paint and thickness applies both to the stroke and the mark itself.

Default: `5pt`

show-labels `bool`

Whether to label the tip/tail origin/end points.

Default: `true`

`show-offsets` `bool`

Whether to visualise the `cap-offset()` values.

Default: `true`

`offset-range` `number`

The span above and below the stroke line to plot the cap offsets, in multiples of the stroke's thickness.

Default: `6`

`shapes.typ`

To use built-in shapes in a diagram, import them with:

```
#import fletcher: shapes
#diagram(node([Hello], stroke: 1pt, shape: shapes.hexagon))
```

or:

```
#import fletcher.shapes: hexagon
#diagram(node([Hello], stroke: 1pt, shape: hexagon))
```

To set a shape parameter, use `shape.with(..)`, for example `hexagon.with(angle: 45deg)`.

- `rect()`
- `circle()`
- `ellipse()`
- `pill()`
- `parallelogram()`
- `diamond()`
- `triangle()`
- `house()`
- `chevron()`
- `hexagon()`
- `octagon()`

`rect()`

The standard rectangle node shape.

A string `"rect"` or the element function `rect` given to the shape option of `node()` are interpreted as this shape.

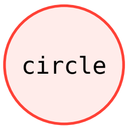
rect

```
rect(node, extrude)
```

circle()

The standard circle node shape.

A string `"circle"` or the element function `circle` given to the `shape` option of `node()` are interpreted as this shape.




```
circle(node, extrude)
```

ellipse()

An elliptical node shape.


```
ellipse(
  node,
  extrude,
  scale: number,
)
```

scale `number`

Scale factor for ellipse radii.

Default: `1`

pill()

A capsule node shape.


```
pill(node, extrude)
```

parallelogram()

A slanted rectangle node shape.


```
parallelogram(
  node,
  extrude,
  angle: angle,
  fit: number,
)
```

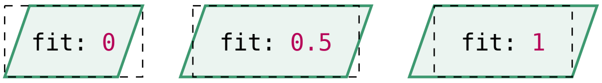
angle `angle`

Angle of the slant, `0deg` is a rectangle. Don't set to `90deg` unless you want your document to be larger than the solar system.

Default: `20deg`

fit `number`

Adjusts how comfortably the parallelogram fits the label.



Default: `0.8`

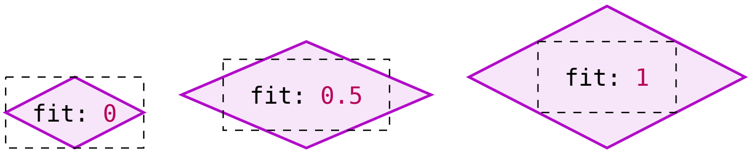
diamond()

A rhombus node shape.


```
diamond(
  node,
  extrude,
  fit: number,
)
```

fit `number`

Adjusts how comfortably the diamond fits the label.

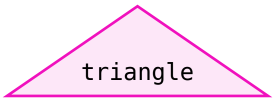


Default: `0.5`

`triangle()`

An isosceles triangle node shape.

One of `angle` or `aspect` may be given, but not both. The triangle's base coincides with the label's base and widens to enclose the label; see https://www.desmos.com/calculator/i4i9svunj4.




```
triangle(
  node,
  extrude,
  dir: top bottom left right,
  angle: angle auto,
  aspect: number auto,
  fit: number,
)
```

`dir` `top` or `bottom` or `left` or `right`

Direction the triangle points.

Default: `top`

`angle` `angle` or `auto`

Angle of the triangle opposite the base.

Default: `auto`

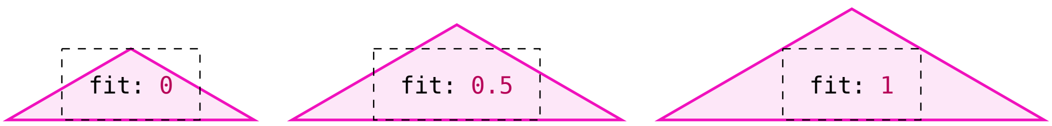
`aspect` `number` or `auto`

Aspect ratio of triangle, or the ratio of its base to its height.

Default: `auto`

`fit` `number`

Adjusts how comfortably the triangle fits the label.



Default: `0.8`

`house()`

A pentagonal house-like node shape.


```
house(
  node,
  extrude,
  dir: top bottom left right ,
  angle: angle ,
)
```

`dir` `top` or `bottom` or `left` or `right`

Direction of the roof of the house.

Default: `top`

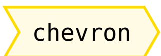
`angle` `angle`

The slant of the roof. Set to `0deg` for a rectangle, and to `90deg` for a document stretching past Pluto.

Default: `10deg`

`chevron()`

A chevron node shape.




```
chevron(
  node,
  extrude,
  dir: top bottom left right ,
  angle: angle ,
  fit: number ,
)
```

`dir` `top` or `bottom` or `left` or `right`

Direction the chevron points.

Default: `right`

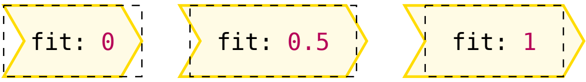
`angle` `angle`

The slant of the roof. Set to `0deg` for a rectangle, and to `90deg` for a document stretching past Pluto.

Default: `30deg`

fit `number`

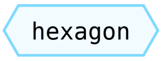
Adjusts how comfortably the chevron fits the label.



Default: `0.8`

hexagon()

An (irregular) hexagon node shape.




```
hexagon(
  node,
  extrude,
  angle: angle,
  fit: number,
)
```

angle `angle`

Half the exterior angle, `0deg` being a rectangle.

Default: `30deg`

fit `number`

Adjusts how comfortably the hexagon fits the label.

Default: `0.8`

octagon()

A truncated rectangle node shape.


```
octagon(
  node,
  extrude,
  truncate: number length,
)
```

truncate `number` or `length`

Size of the truncated corners. A number is interpreted as a multiple of the smaller of the node's width or height.

Default: `0.5`

coords.typ

- `uv-to-xy()`
- `xy-to-uv()`
- `duv-to-dxy()`
- `dxy-to-duv()`
- `vector-polar-with-xy-or-uv-length()`
- `resolve-label-coordinate()`
- `resolve-relative-coordinates()`

`uv-to-xy()`

Convert from elastic to absolute coordinates, $(u, v) \mapsto (x, y)$.

Elastic coordinates are specific to the diagram and adapt to row/column sizes; *absolute* coordinates are the final, physical lengths which are passed to `cetz`.

`uv-to-xy(grid: dictionary, uv: array)`

grid `dictionary`

Representation of the grid layout, including:

- `origin`
- `centers`
- `spacing`
- `flip`

The `grid` is passed to the `render` option of `diagram()`.

uv `array`

Elastic coordinate, `(float, float)`.

`xy-to-uv()`

Convert from absolute to elastic coordinates, $(x, y) \mapsto (u, v)$.

Inverse of `uv-to-xy()`.

`xy-to-uv(grid, xy)`

duv-to-dxy()

Jacobian of the coordinate map uv-to-xy().

Used to convert a "nudge" in uv coordinates to a "nudge" in xy coordinates. This is needed because uv coordinates are non-linear (they're elastic). Uses a balanced finite differences approximation.

```
duv-to-dxy(
  grid: dictionary,
  uv: array,
  duv: array,
)
```

grid `dictionary`

Representation of the grid layout. The `grid` is passed to the `render` option of `diagram()`.

uv `array`

The point `(float, float)` in the uv-manifold where the shift tangent vector is rooted.

duv `array`

The shift tangent vector `(float, float)` in uv coordinates.

dxy-to-duv()

Jacobian of the coordinate map xy-to-uv().

```
dxy-to-duv(
  grid,
  xy,
  dxy,
)
```

vector-polar-with-xy-or-uv-length()

Return a vector in xy coordinates with a given angle θ in xy-space but with a length specified in either xy-space or uv-space.

```
vector-polar-with-xy-or-uv-length(
  grid,
  xy,
  target-length,
  θ,
)
```

`resolve-label-coordinate()`

Convert labels into the coordinates of a node with that label, leaving anything else unchanged.

```
resolve-label-coordinate(nodes, coord)
```

`resolve-relative-coordinates()`

Given a sequence of coordinates of the form `(x, y)` or `(rel: (Δx, Δy))`, return a sequence in the form `(x, y)` where relative coordinates are applied relative to the previous coordinate in the sequence.

The first coordinate must be of the form `(x, y)`.

```
resolve-relative-coordinates(coords)
```

`layout.typ`

- `compute-node-sizes()`
- `compute-node-enclosures()`
- `expand-fractional-rects()`
- `interpret-axes()`
- `compute-cell-sizes()`
- `compute-cell-centers()`
- `compute-grid()`
- `apply-edge-shift()`

`compute-node-sizes()`

Measure node labels with the style context and resolve node shapes.

Widths and heights that are `auto` are determined by measuring the size of the node's label.

```
compute-node-sizes(nodes, styles)
```

`compute-node-enclosures()`

Process the `enclose` options of an array of nodes.

```
compute-node-enclosures(nodes, grid)
```

`expand-fractional-rects()`

Convert an array of rects `(center: (x, y), size: (w, h))` with fractional positions into rects with integral positions.

If a rect is centered at a factional position `floor(x) < x < ceil(x)`, it will be replaced by two new rects centered at `floor(x)` and `ceil(x)`. The total width of the original rect is split across the two new rects according two which one is closer. (E.g., if the original rect is at `x = 0.25`, the new rect at `x = 0` has 75% the original width and the rect at `x = 1` has 25%.) The same splitting procedure is done for `y` positions and heights.

```
expand-fractional-rects(rects: array) -> array
```

`rects` `array`

An array of rects of the form `(center: (x, y), size: (width, height))`. The coordinates `x` and `y` may be floats.

`interpret-axes()`

Interpret the `axes` option of `diagram()`.

Returns a dictionary with:
- `x`: Whether u is reversed
- `y`: Whether v is reversed
- `xy`: Whether the axes are swapped

```
interpret-axes(axes: array) -> dictionary
```

`axes` `array`

Pair of directions specifying the interpretation of (u, v) coordinates. For example, `(ltr, ttb)` means u goes $\rightarrow$ and v goes $\downarrow$.

`compute-cell-sizes()`

Determine the number and sizes of grid cells needed for a diagram with the given nodes and edges.

Returns a dictionary with:
- `origin: (u-min, v-min)` Coordinate at the grid corner where elastic/`uv` coordinates are minimised.
- `cell-sizes: (x-sizes, y-sizes)` Lengths and widths of each row and column.

```
compute-cell-sizes(
  grid: dictionary,
  nodes,
  edges,
)
```

grid `dictionary`

Representation of the grid layout, including:
- `flip`

`compute-cell-centers()`

Determine the centers of grid cells from their sizes and spacing between them.

Returns the a dictionary with:
- `centers: (x-centers, y-centers)` Positions of each row and column, measured from the corner of the bounding box.
- `bounding-size: (x-size, y-size)` Dimensions of the bounding box.

```
compute-cell-centers(grid: dictionary) -> dictionary
```

grid `dictionary`

Representation of the grid layout, including:
- `cell-sizes: (x-sizes, y-sizes)` Lengths and widths of each row and column.
- `spacing: (x-spacing, y-spacing)` Gap to leave between cells.

`compute-grid()`

Determine the number, sizes and relative positions of rows and columns in the diagram's coordinate grid.

Rows and columns are sized to fit nodes. Coordinates are not required to start at the origin, `(0,0)`.

```
compute-grid(
  nodes,
  edges,
  options,
)
```

`apply-edge-shift()`

Apply the `shift` option of `edge()` by translating edge vertices.

```
apply-edge-shift(grid: dictionary, edge: dictionary)
```

grid `dictionary`

Representation of the grid layout. This is needed to support shifts specified as coordinate lengths.

edge `dictionary`

The edge with a `shift` entry.

draw.typ

draw-edge-line()

Draw a straight edge.

`draw-edge-line(edge: dictionary, debug: int)`

edge `dictionary`

The edge object, a dictionary, containing:
- `vertices`: an array of two points, the line's start and end points.
- `extrude`: An array of extrusion lengths to apply a multi-stroke effect with.
- `stroke`: The stroke style.
- `marks`: An array of marks to draw along the edge.
- `label`: Content for label.
- `label-side`, `label-pos`, `label-sep`, and `label-anchor`.

debug `int`

Level of debug details to draw.

Default: `0`

draw-edge-arc()

Draw a bent edge.

`draw-edge-arc(edge: dictionary, debug: int)`

edge `dictionary`

The edge object, a dictionary, containing:
- `vertices`: an array of two points, the arc's start and end points.
- `bend`: The angle of the arc.
- `extrude`: An array of extrusion lengths to apply a multi-stroke effect with.
- `stroke`: The stroke style.
- `marks`: An array of marks to draw along the edge.
- `label`: Content for label.
- `label-side`, `label-pos`, `label-sep`, and `label-anchor`.

debug `int`

Level of debug details to draw.

Default: `0`

`draw-edge-polyline()`

Draw a multi-segment edge

```
draw-edge-polyline(edge: dictionary , debug: int )
```

edge `dictionary`

The edge object, a dictionary, containing:
- `vertices`: an array of at least two points to draw segments between.
- `corner-radius`: Radius of curvature between segments.
- `extrude`: An array of extrusion lengths to apply a multi-stroke effect with.
- `stroke`: The stroke style.
- `marks`: An array of marks to draw along the edge.
- `label`: Content for label.
- `label-side`, `label-pos`, `label-sep`, and `label-anchor`.

debug `int`

Level of debug details to draw.

Default: `0`

`find-farthest-intersection()`

Of all the intersection points within a set of CeTZ objects, find the one which is farthest from a target point and pass it to a callback.

If no intersection points are found, use the target point itself.

```
find-farthest-intersection(
  objects: cetz array none ,
  target: point ,
  callback,
)
```

objects `cetz array` or `none`

Objects to search within for intersections. If `none`, callback is immediately called with `target`.

target `point`

Target point to sort intersections by proximity with, and to use as a fallback if no intersections are found.

`get-node-anchor()`

Get the anchor point around a node outline at a certain angle.

```
get-node-anchor(
  node,
  θ,
  callback,
)
```

`defocus-adjustment()`

Return the anchor point for an edge connecting to a node with the "defocus" adjustment.

Basically, for very long/wide nodes, don't make edges coming in from all angles go to the exact node center, but "spread them out" a bit.

See https://www.desmos.com/calculator/irt0mvixky.

```
defocus-adjustment(node, θ)
```

`draw-debug-axes()`

Draw diagram coordinate axes.

```
draw-debug-axes(grid: dictionary , debug)
```

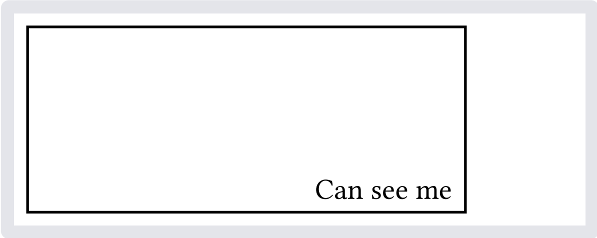
grid `dictionary`

Dictionary specifying the diagram's grid, containing:
- `origin: (u-min, v-min)`, the minimum values of elastic coordinates,
- `flip: (x, y, xy)`, the axes orientation (see `interpret-axes()`),
- `centers: (x-centers, y-centers)`, the physical offsets of each row and each column,
- `cell-sizes: (x-sizes, y-sizes)`, the physical sizes of each row and each column.

hide()

Make diagram contents invisible, with or without affecting layout. Works by wrapping final drawing objects in `cetz.draw.hide`.

```
rect(diagram({
  fletcher.hide({
    node((0,0), [Can't see me])
    edge("->")
  })
  node((1,1), [Can see me])
}))
```



`hide(objects: content array, bounds: bool)`

`objects` `content` or `array`

Diagram objects to hide.

`bounds` `bool`

If `false`, layout is as if the objects were never there; if `true`, the layout treats the objects is present but invisible.

Default: `true`

utils.typ

- interp()
- interp-inv()
- get-arc-connecting-points()
- is-space()

interp()

Linearly interpolate an array with linear behaviour outside bounds

```
interp(
  values: array,
  index: int float,
  spacing: length,
)
```

`values` `array`

Array of lengths defining interpolation function.

`index` `int` or `float`

Index-coordinate to sample.

spacing length

Gradient for linear extrapolation beyond array bounds.

Default: `0pt`

`interp-inv()`

Inverse of `interp()`.

```
interp-inv(
  values: array ,
  value,
  spacing: length ,
)
```

values array

Array of lengths defining interpolation function.
- value: Value to find the interpolated index of.

spacing length

Gradient for linear extrapolation beyond array bounds.

Default: `0pt`

`get-arc-connecting-points()`

Determine arc between two points with a given bend angle

The bend angle is the angle between chord of the arc (line connecting the points) and the tangent to the arc and the first point.

Returns a dictionary containing:
- `center`: the center of the arc's curvature
- `radius`
- `start`: the start angle of the arc
- `stop`: the end angle of the arc

```
get-arc-connecting-points(
  from: point ,
  to: point ,
  angle: angle ,
) -> dictionary
```

from point

2D vector of initial point.

to `point`

2D vector of final point.

angle `angle`

The bend angle between chord of the arc (line connecting the points) and the tangent to the arc and the first point.

is-space()

Return true if a content element is a space or sequence of spaces

```
is-space(el)
```