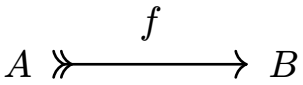


fletcher

(noun) a maker of arrows

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

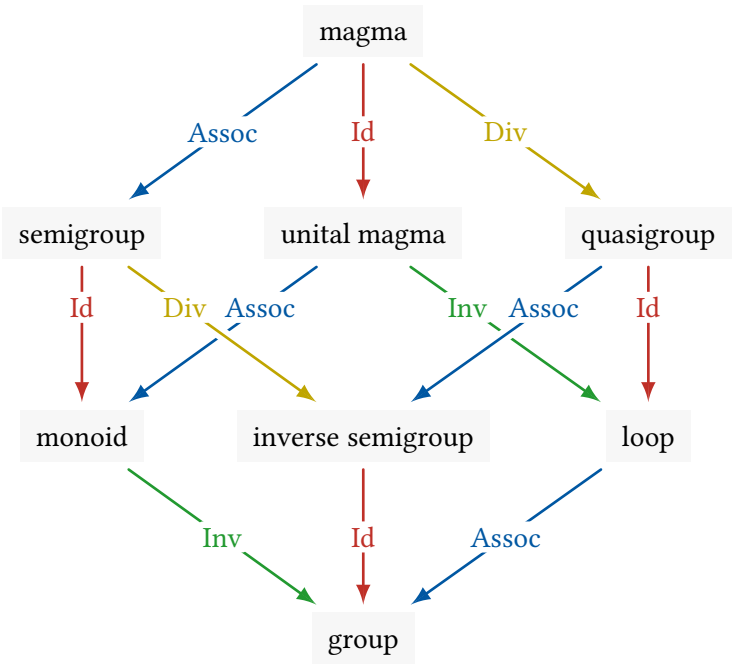
Commutative diagrams, flow charts, state machines, block diagrams...

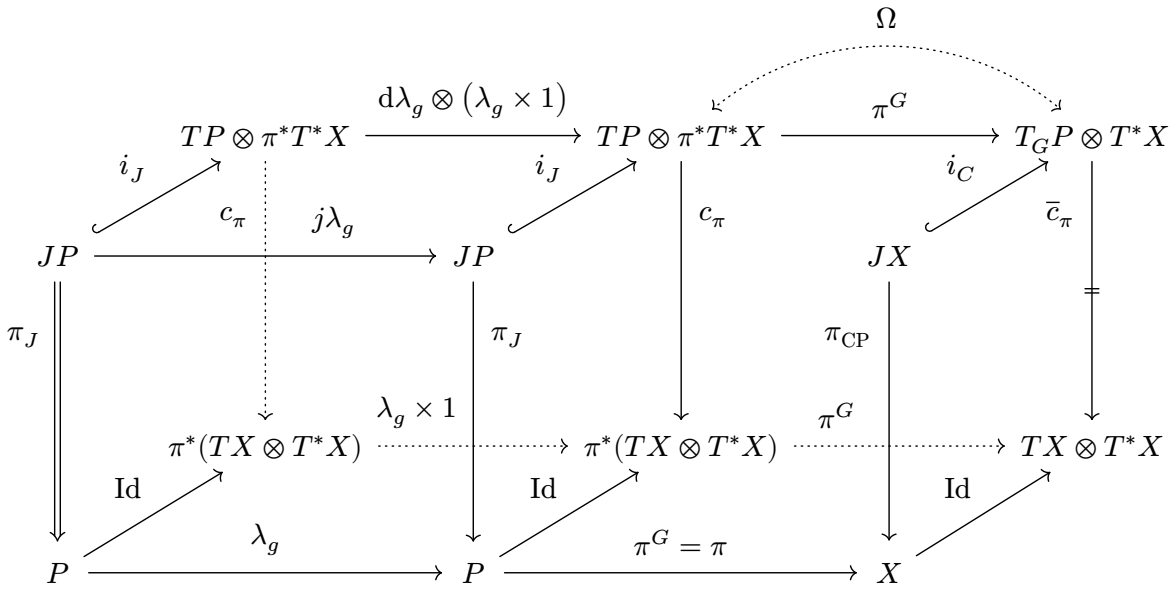
`github.com/Jollywatt/typst-fletcher`

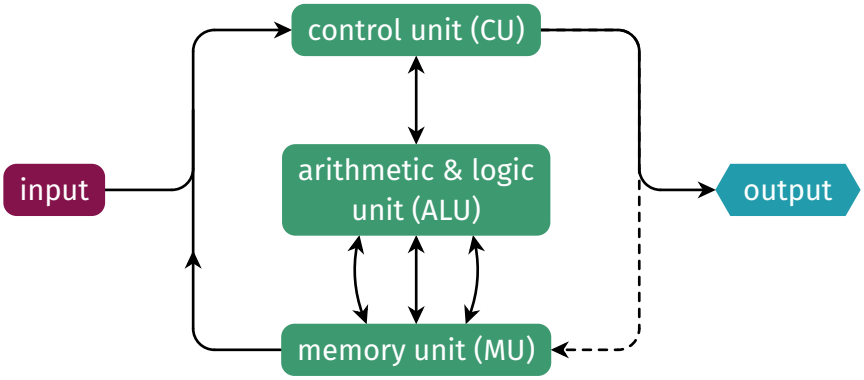
Version 0.5.8

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
$T_GP \otimes T^*X$
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

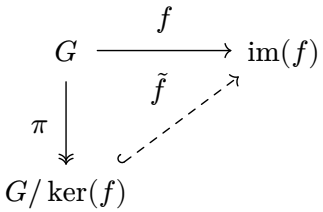

control unit (CU)
input
arithmetic & logic unit (ALU)
output
memory unit (MU)

Usage examples

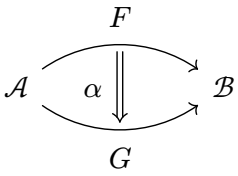
Avoid importing everything with * as many internal functions are also exported.

```
#import "@preview/fletcher:0.5.8" as fletcher: diagram, node, edge
```

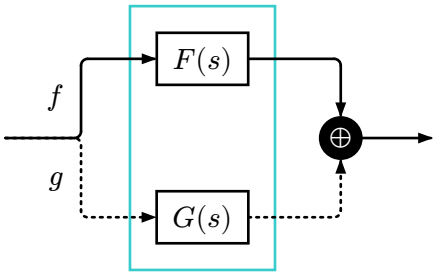
```
// You can specify nodes in math-mode, separated by `&`:
#diagram($
  G edge(f, ->) edge("d", pi, ->>) & im(f) \
  G slash ker(f) edge("ur", tilde(f), "hook-->")
$)
```




```
// 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), $alpha$, "=>")
})
```



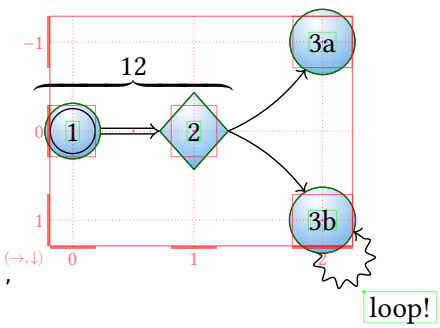

```
#diagram(
  spacing: (10mm, 5mm), // wide columns, narrow rows
  node-stroke: 1pt,     // outline node shapes
  edge-stroke: 1pt,     // make lines thicker
  mark-scale: 60%,      // make arrowheads smaller
  edge((-2,0), "r,u,r", "-|>", $f$, label-side: left),
  edge((-2,0), "r,d,r", "..|>", $g$),
  node((0,-1), $F(s)$),
  node((0,+1), $G(s)$),
  node(enclose: ((0,-1), (0,+1)), stroke: teal, inset: 10pt,
       snap: false), // prevent edges snapping to this node
  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($A edge(->, text(#0.8em, f)) & B$).
```

An equation $f : A \to B$ and an inline diagram $A \xrightarrow{f} B$.

```
#import fletcher.shapes: diamond, brace
#diagram(
  debug: 3,
  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
  edge("=>"),
  node((1,0), [2], name: <2>, shape: diamond),
  node((2,-1), [3a], name: <3a>),
  node((2,+1), [3b], name: <3b>),
  node(enclose: (<1>, <2>), shape: brace.with(dir: top, label: [12])),
  edge(<2.east>, "->", <3a>, bend: -15deg),
  edge(<2.east>, "->", <3b>, bend: +15deg),
  edge(<3b>, "~>", <3b>, bend: -130deg, loop-angle: 120deg)[loop!],
)
```



Diagrams

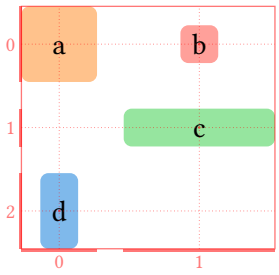
Diagrams are collections of *nodes* and *edges* rendered on a CeTZ canvas with `diagram()`.

Elastic coordinates

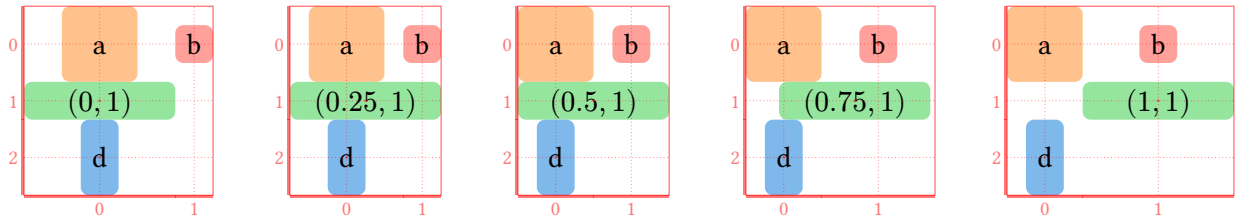
Diagrams are laid out on a *flexible coordinate grid*, visible when the `debug` option of `diagram()` is on. When a node is placed, the rows and columns grow to accommodate the node's size, like a table.

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: 1,
  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),
)
```



So far, this is just like a table — however, elastic coordinates can be *fractional*. Notice how the column sizes change as the green node is gradually moved between columns:



Absolute coordinates

As well as *elastic* or uv coordinates, which are row/column numbers, you can also use *absolute* or xy coordinates, which are physical lengths.

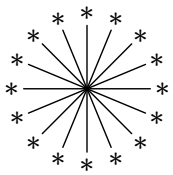
Elastic coordinates, e.g., `(2, 1)`	**Physical coordinates, e.g., `(10mm, 5mm)`**
Dimensionless, dependent on row/column sizes.	Lengths, independent of row/column sizes.
Node positions and sizes affect diagram layout.	Nodes are *floating* and never affect layout.

Absolute coordinates let you position nodes *exactly*, whereas elastic coordinates are useful for table-like layouts. Absolutely positioned nodes never affect the positions of other nodes — the row and column sizes of a diagram depend only on the positions and sizes of nodes at elastic coordinates.

Coordinate expressions

You can use CeTZ-style coordinate expressions such as *relative* `(rel: (1, 2))`, *polar* `(45deg, 1cm)`, *interpolating* `(<P>, 80%, <Q>)`, *perpendicular* `(<X>, "|-", <Y>)`, and so on.

```
#diagram(
  node((1, 0), name: <origin>), // elastic coordinate
  for θ in range(16).map(i => i/16*360deg) {
    node((rel: (θ, 10mm), to: <origin>), $ * $, inset: 1pt) // absolute offset
    edge(<origin>)
  }
)
```

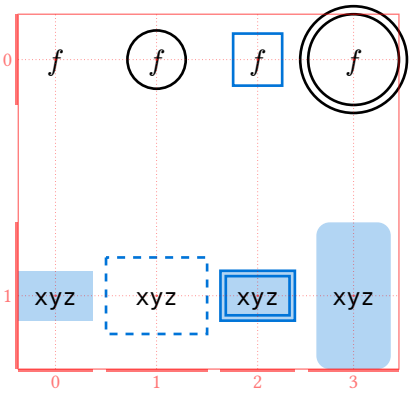


Nodes

`node(coord, label, ..options)`

Nodes are content centered at a particular coordinate. 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:

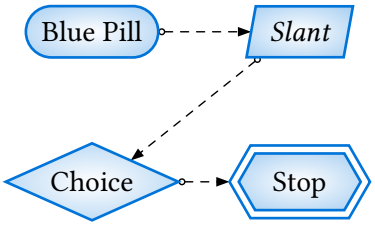
```
#diagram(
  debug: true, // show a coordinate grid
  spacing: (5pt, 4em), // small column gaps, large row spacing
  node((0,0), $f$),
  node((1,0), $f$, stroke: 1pt),
  node((2,0), $f$, stroke: blue, shape: rect),
  node((3,0), $f$, stroke: 1pt, radius: 6mm, extrude: (0, 3)),
  {
    let b = blue.lighten(70%)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



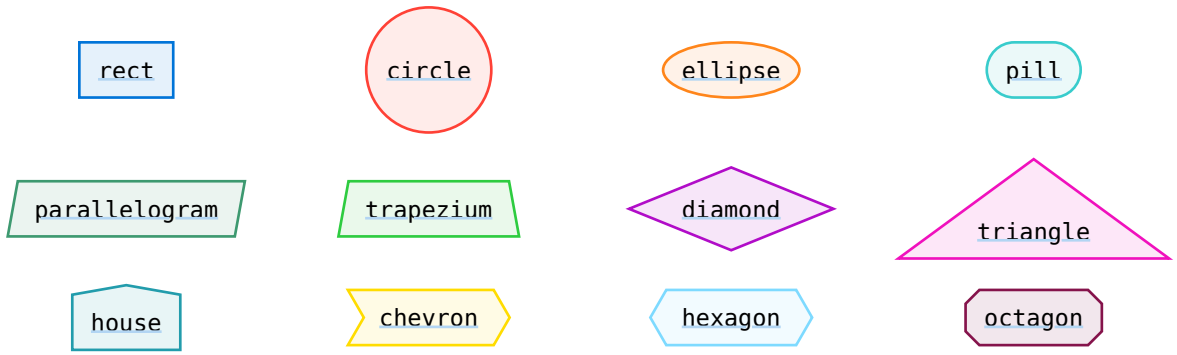
Node shapes

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
#diagram(
  node-fill: gradient.radial(white, blue, radius: 200%),
  node-stroke: blue,
  (
    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. The predefined shapes are:

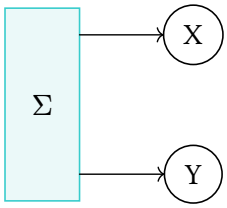


Shapes respect the `stroke`, `fill`, `width`, `height`, and `extrude` options of `node()`. There are also node "shapes" for placing a `stretched-glyph()` along the edge of a nodes, especially useful with `enclose`.

Node groups

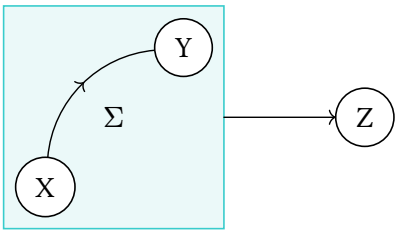
Nodes are usually centered at a particular coordinate, but they can also `enclose` multiple centers. When the `enclose` option of `node()` is given, the node automatically resizes.

```
#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:

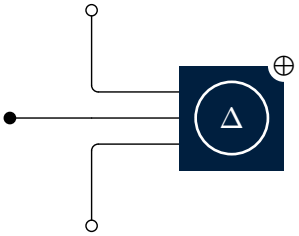
```
#diagram(
  node-stroke: 0.6pt,
  node-fill: white,
  node((0,1), [X]),
  edge("->-", bend: 40deg),
  node((1,0), [Y], name: <y>),
  node($Sigma$, enclose: ((0,1), <y>),
       stroke: teal, fill: teal.lighten(90%),
       snap: -1, // prioritise other nodes when auto-snapping
       name: <group>),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



Node anchors

You can reference anchor points on node shapes like in CeTZ, provided the node has a `name`. For example, `<A.north>` and `(name: "A", anchor: "north")` are equivalent coordinate expressions that can be referenced in other nodes or edges.

```
#diagram(
  node-shape: rect,
  node(circle(stroke: white, text(white, $Delta$)), name: <A>, fill: navy),
  node(<A.north-east>, circle(fill: white, radius: 6pt, $ plus.circle $)),
  edge((<A.north-west>, 25%, <A.south-west>), "l,u", "-O"),
  edge((<A.north-west>, 50%, <A.south-west>), "l,l", "-@"),
  edge((<A.north-west>, 75%, <A.south-west>), "l,d", "-O"),
)
```



Node anchors count as *absolute* coordinates, meaning that nodes positioned with anchors are *floating* and never affect row and column sizes.

Edges

`edge(..vertices, marks, label, ..options)`

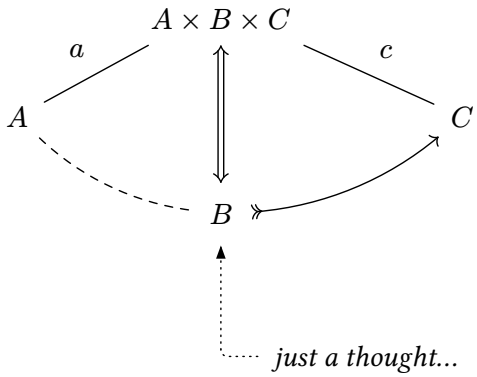
An edge connects two coordinates. By default, edges *snap* to nodes' bounding shapes (after applying the node's `outset`). This can be adjusted with the `snap-to` option of `edge()`.

An edge 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)
})
```

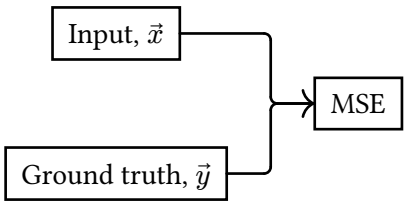


Specifying edge vertices

The first few arguments given to `edge()` specify its `vertices`, of which there can be two or more. Like node positions, vertices may be CeTZ-style coordinate expressions, combining elastic and physical coordinates, and node anchors.

Here is a more advanced example using coordinate expressions and `()`, the edge's previous vertex.

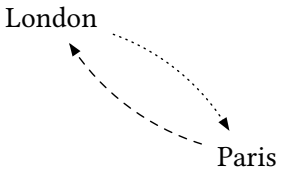
```
#diagram(edge-stroke: 1pt, node-stroke: 1pt, {
  node((0,0), name: <x>)[Input, $arrow(x)$]
  node((0,1), name: <y>)[Ground truth, $arrow(y)$]
  node((1,0.5), name: <out>)[MSE]
  let verts = ( // () means the previous vertex
    (), "-|", (<y.east>, 50%, <out.west>)),
    (), "|-", <out>), <out>)
  edge(<x>, ..verts, "->") // () == <x>
  edge(<y>, ..verts)  // () == <y>
})
```



Use `auto` for the previous or next node

If an edge's first or last vertex is `auto`, the previous or next node is used, according to the order that nodes and edges are passed to `diagram()`. A single vertex, such as `edge(to)`, is interpreted as `edge(auto, to)`. Given no vertices, an edge connects the nearest nodes on either side.

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



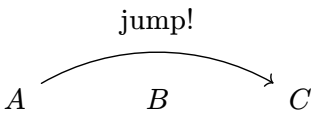
Implicit coordinates can be handy for diagrams in math-mode:

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

Relative coordinate shorthands

You may use strings such as `"u"` for up or `"sw"` for south west as shorthands for relative vertex coordinates of the form `(rel: (du, dv))`. Any combination of **t**op/**u**p/**n**orth, **b**ottom/**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 $)
```



Node anchors

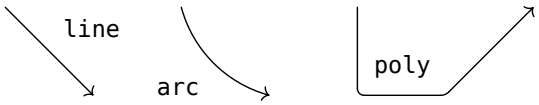
Nodes can be given a `name`, which is a label (not a string) identifying that node. A label as an edge vertex is interpreted as the position of the node with that label.

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

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 coordinate shorthands (see above), so that in the example above, the `"poly"` edge could also be written in these equivalent ways:

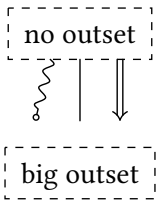
```
edge((4,0), (rel: (0,1)), (rel: (1,0)), (rel: (1,-1)), "->", `poly`)
edge((4,0), "d", "r", "ur", "->", `poly`) // using relative coordinate names
edge((4,0), "d,r,ur", "->", `poly`) // shorthand
```

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

Ways to adjust edge connection points

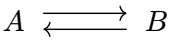
A node's `outset` controls how *close* edges connect to the node's boundary. To adjust *where* along the boundary the edge connects, you can adjust the edge's end coordinates by a fractional amount.

```
#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.

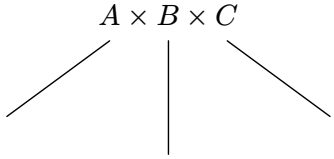


Figure 1: Node with defocus (default)

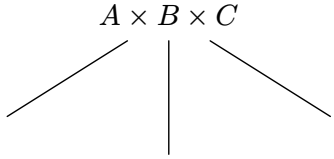
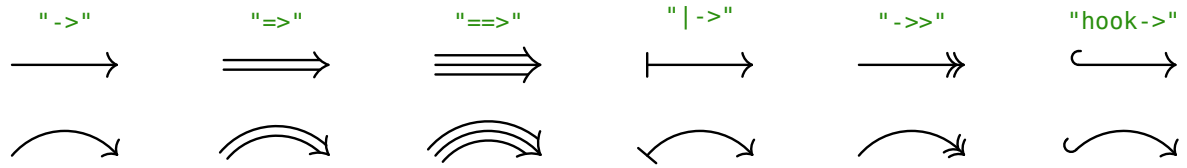


Figure 2: No defocus adjustment

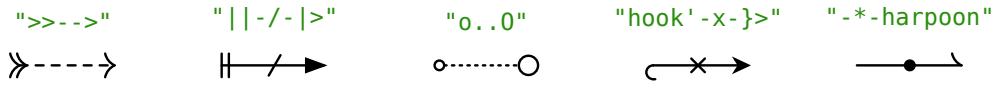
The strength of this adjustment is controlled by the `defocus` option of `node()` or the `node-defocus` option of `diagram()`.

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 →, ⇒, ⇛, ↦, ↠, and ↪ 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 (see Table 1) are defined in the state variable `fletcher.MARKS`, which you may access with `context fletcher.MARKS.get()`. You add or tweak mark styles by modifying `fletcher.MARKS`, as described in Mark objects.

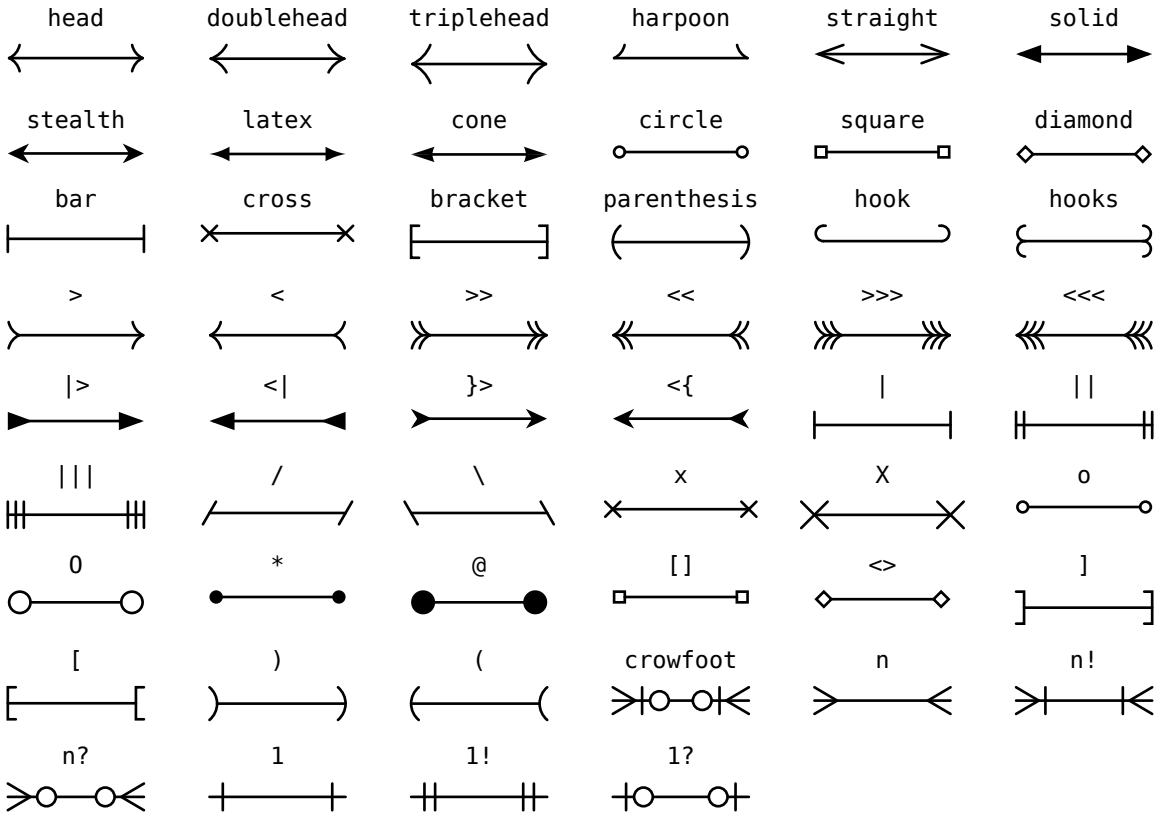
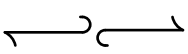


Table 1: Default marks by name. Properties such to size, angle, spacing, or fill can be adjusted.

Marks can be flipped by appending `'` to the name.

```
#diagram(edge("harpoon'-hook", stroke: 1pt))
#diagram(edge("hook'-harpoon", stroke: 1pt))
```



If there is a common mark style that you believe should be included with `fletcher` by default, please open an issue!

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: 28mm,
  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. These CeTZ objects are translated and scaled to fit the edge; the mark should be centered at `(0, 0)`, and the stroke's thickness is defined as the unit length. For example, here is a basic circle mark:

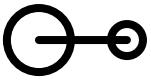
```
#import cetz.draw
#let my-mark = (
  draw: draw.circle((0,0), radius: 2, fill: none)
)
#diagram(
  edge((0,0), (1,0), stroke: 1pt, marks: (my-mark, my-mark), bend: 30deg),
  edge((0,1), (1,1), stroke: 3pt + orange, marks: (none, my-mark)),
)
```

A mark object can contain arbitrary parameters. Parameters can be functions `mark => (..)` referencing other `mark` parameters defined earlier. 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)
)
```

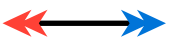
This form makes it easier to change the size without modifying the `draw` function, for example:

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



Lastly, mark objects may *inherit* properties from other marks in `fletcher.MARKS` by containing an `inherit` entry, for example:

```
#let my-mark = (
  inherit: "stealth", // base mark on `fletcher.MARKS.stealth`
  fill: red,
  stroke: none,
  extrude: (0, -3),
)
#diagram(edge("rr", stroke: 2pt, marks: (my-mark, my-mark + (fill: blue))))
```



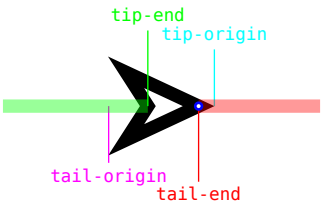
Internally, marks are passed to `resolve-mark()`, which resolves all entries to their final values.

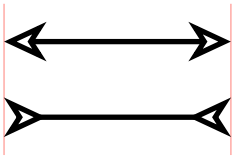
Special mark properties

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 `extruded` 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:





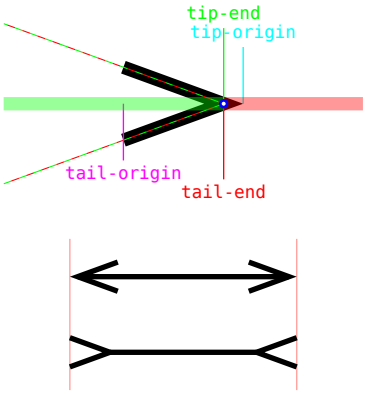
See `mark-debug()` and `cap-offset()` for details.

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)
```



Defining 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:

```
Original marks:
#diagram(spacing: 2cm, edge("<->", stroke: 1pt))

#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)),
    ),
  ),
))

Updated marks:
#diagram(spacing: 2cm, edge("multi->-multi", stroke: 1pt + eastern))
```

Original marks: ⟷

Updated marks: ≫→≪

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:

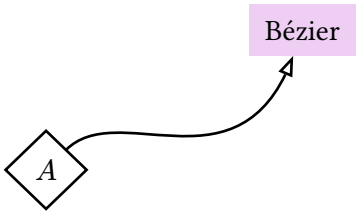
```
#diagram(
  node((0,1), $A$, stroke: 1pt, shape: fletcher.shapes.diamond),
  node((2,0), [Bézier], fill: purple.lighten(80%)),

  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
          )
        })
      })
    })
  }
)
```

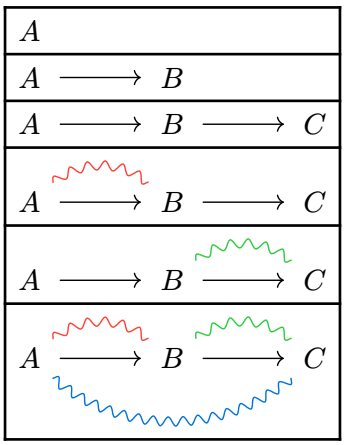


Touying integration

You can create incrementally-revealed diagrams with Touying presentation slides by defining a `touying-reducer`. You must redefine `diagram` to use this reducer so that Touying primitives like `pause`, `uncover`, `only`, and so on are understood. For example, here is a simple animated diagram:

```
#import "@preview/touying:0.5.5": *
#show: themes.simple.simple-theme.with(aspect-ratio: "16-9")
#let diagram = touying-reducer.with(
  reduce: fletcher.diagram, cover: fletcher.hide)

#slide(repeat: 6, self => {
  let (uncover, only, alternatives) = utils.methods(self)
  diagram(
    node((0, 0), name: <A>)[$A$],
    pause,
    edge("->"),
    node((1, 0), name: <B>)[$B$],
    pause,
    edge("->"),
    node((2, 0), name: <C>)[$C$],
    only("4,6", edge(<A>, "~", <B>, bend: 40deg, stroke: red)),
    only("5,6", edge(<B>, "~", <C>, bend: 40deg, stroke: green)),
    only("6", edge(<C>, "~", <A>, bend: 40deg, stroke: blue)),
  )
})
```



Reference

Main functions

`diagram()`

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

```
diagram(
  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-shape: rect circle function,
  node-fill: paint,
  node-defocus: number,
  label-sep: length,
  label-size: length,
  label-wrapper: function,
  mark-scale: percent,
  crossing-fill: paint,
  crossing-thickness: number,
  render: function,
  ..args: array,
)
```

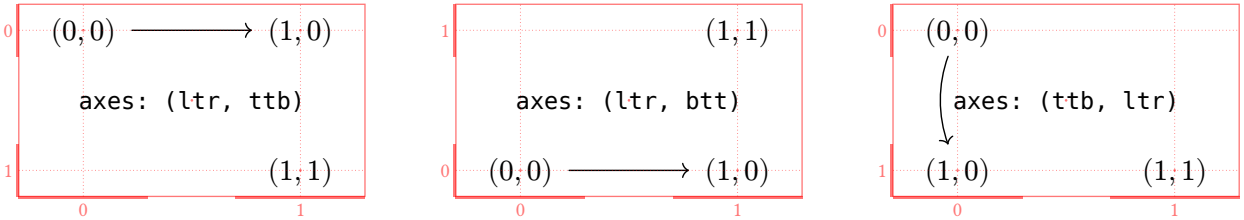
`debug` `bool` or `1` or `2` or `3` default `false`

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

`axes` `pair of directions` default `(ltr, ttb)`

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)`.



spacing `length` or `pair of lengths` default `3em`

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)`.

cell-size `length` or `pair of lengths` default `0pt`

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

edge-stroke `stroke` default `0.048em`

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`.

node-stroke `stroke` or `none` default `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`.

edge-corner-radius `length` or `none` default `2.5pt`

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

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

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

node-inset `length` or `pair of lengths` default `6pt`

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

node-outset `length` or `pair of lengths` default `0pt`

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

node-shape `rect` or `circle` or `function` default `auto`

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

`node-fill` `paint` default `none`

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

`node-defocus` `number` default `0.2`

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

`label-sep` `length` default `0.4em`

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

`label-size` `length` default `1em`

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

`label-wrapper` `function`

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

Default:
```
edge => box(
    [#edge.label],
    inset: .2em,
    radius: .2em,
    fill: edge.label-fill,
  )
```

`mark-scale` `percent` default `100%`

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

`crossing-fill` `paint` default `white`

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

`crossing-thickness` `number` default `5`

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

`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))
  }`

`..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
$)
```

node()

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

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

pos `coordinate` default `auto`

Position of the node, or its center coordinate. This may be an elastic (row/column) coordinate like `(2, 1)`, or a CeTZ-style coordinate expression like `(rel: (30deg, 1cm), to: (2, 1))`.

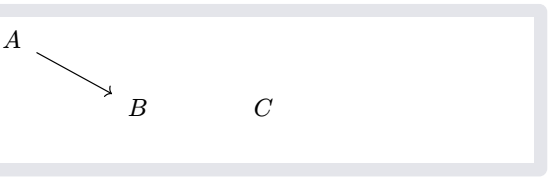
See the options of `diagram()` to control the physical scale of elastic coordinates.

name `label` or `string` or `none` default `none`

An optional name to give the node.

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

```
#diagram(
  node((0,0), $A$, name: <A>),
  node((1,0.6), $B$, name: <B>),
  edge(<A>, <B>, "->"),
  node((rel: (1, 0), to: <B>), $C$)
)
```



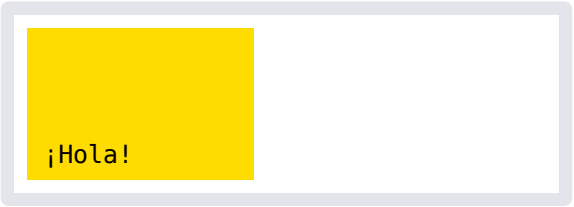
Node names are *labels* (instead of strings like in CeTZ) to disambiguate them from other positional string arguments given to `edge()`. If a string is given, it is converted. (Since these labels are never inserted into the final document, they cannot interfere with other document labels.)

label `content` default `none`

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),
)
```

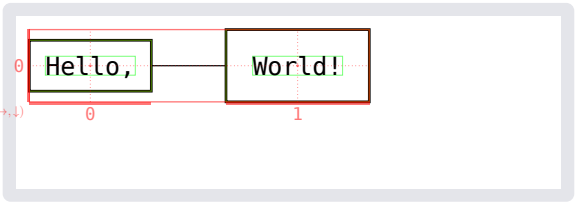


inset `length` default `auto`

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),
)
```



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

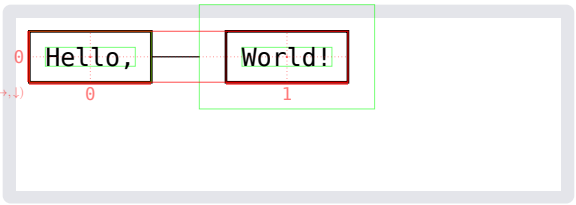
outset `length` default `auto`

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),
)
```



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

fill `paint` default `auto`

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()`.

`stroke` `stroke` default `auto`

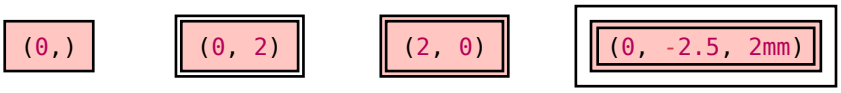
Stroke style for the node outline.

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

`extrude` `array` default `(0,)`

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()`.

`width` `length` or `auto` default `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`.

`height` `length` or `auto` default `auto`

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

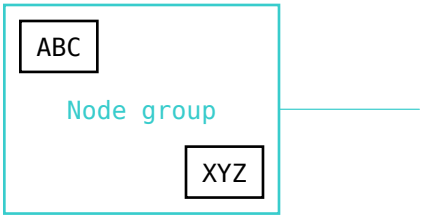
If the height is not `auto`, you can use `align` to control the placement of the node's `label`.

`enclose` `array` default `()`

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),
)
```



`corner-radius` `length` default `auto`

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

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

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

Shape of the node's outline. 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 `rect`, `circle`, `ellipse`, `pill`, `parallelogram`, `trapezium`, `diamond`, `triangle`, `house`, `chevron`, `hexagon`, `octagon`, `brace`, `bracket`, `paren`, `cylinder`, and `database`.

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.

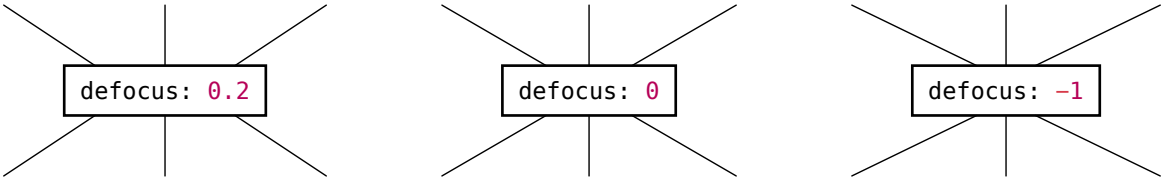
See the `src/shapes.typ` source file for example shape implementations.

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

defocus `number` default `auto`

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()`.

snap `number` or `false` default `0`

The snapping priority for edges connecting to this node. A higher priority means edges will automatically snap to this node over other overlapping nodes. If `false`, edges only snap to this node if manually set with the `snap-to` option of `edge()`.

Setting a lower value is useful if the node `encloses` other nodes that you want to snap to first.

layer `number` default `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()`.

Defaults to layer `0` unless the node `encloses` points, in which case `layer` defaults to `-1`.

post `function` default `x => x`

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.

..args `any`

The first positional argument is `pos` and the second, if given, is `label`.

edge()

Draw a connecting edge in a diagram.

```
edge(
  vertices: array,
  label: content,
  label-side: left right center,
  label-pos: float ratio relative length array,
  label-sep: length,
  label-angle: angle left right top bottom auto,
  label-anchor: anchor,
  label-fill: bool paint,
  label-size: auto length,
  label-wrapper: auto function,
  stroke: stroke,
  dash: string,
  decorations: none string function,
  extrude: array,
  shift: length number pair,
  kind: string,
  bend: angle,
  loop-angle: 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,
  floating: bool,
  post: function,
  ..args: any,
)
```

vertices `array` default `()`

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"`.

label `content` default `none`

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).

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

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

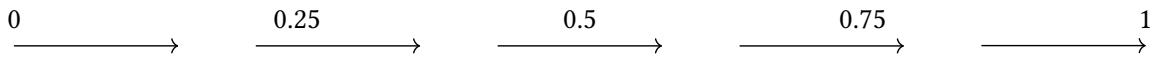
If `center`, then the label is placed directly on the edge and `label-fill` defaults to `true`. When `auto`, a value of `left` or `right` is automatically chosen so that the label is:

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

label-pos `float` or `ratio` or `relative length` or `array` default `50%`

Position of the label along the edge, from the start to end.

A number or ratio between zero and one is interpreted as a fraction of the edge length. Physical and relative relative lengths work too. For example, `100% - 1em` means `1em` from the end.



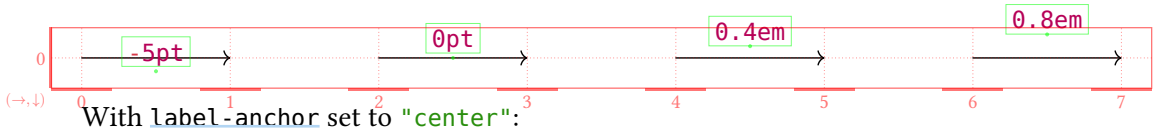
For `"poly"` edges (see Edge types), a number does not specify a fraction of the path length; instead, the kth vertex is at position $\frac{k}{n}$ where n is the number of vertices. Each midpoint is then at $\frac{k}{n} + 0.5$.

As an alternative, the position can be specified by an array `(segment, position)`, where `segment` is an integer that indicates which segment the label is placed on (segment s is the one between vertices s and $s + 1$). `position` is then relative to the specified segment; the segment midpoint is at `(s, 50%)`.

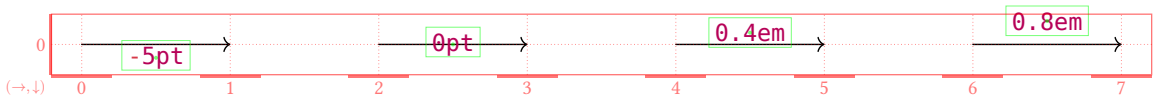
label-sep `length`

Separation between the connector and the label anchor.

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



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



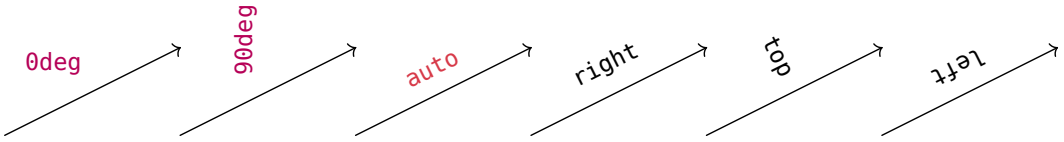
Set `debug` to `2` or higher to see label anchors and outlines as seen here.

Default: the `label-sep` option of `diagram()`

label-angle `angle` or `left` or `right` or `top` or `bottom` or `auto` default `0deg`

Angle to rotate the label (counterclockwise).

If a direction is given, the label is rotated so that the edge travels in that direction relative to the label. If `auto`, the best of `right` or `left` is chosen.



label-anchor `anchor` default `auto`

The CeTZ-style anchor point of the label to use for placement (e.g., `"north-east"` or `"center"`). If `auto`, the best anchor is chosen based on `label-side`, `label-angle`, and the edge's direction.

label-fill `bool` or `paint` default `auto`

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`).

label-size `auto` or `length`

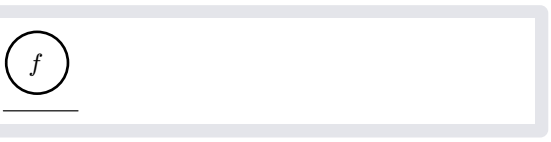
The default text size to apply to edge labels.

Default: the `label-size` option of `diagram()`

label-wrapper `auto` or `function`

Callback function accepting a node dictionary and returning the label content. This is used to add a label background (see `crossing-fill`), and can be used to adjust the label's padding, outline, and so on.

```
#diagram(edge($f$, label-wrapper: e =>
  circle(e.label, fill: e.label-fill)))
```



Default: the `label-wrapper` option of `diagram()`

stroke `stroke` default `auto`

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

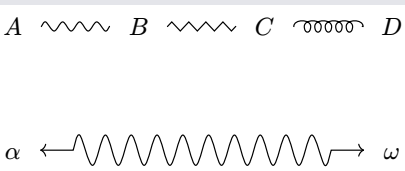
dash `string` default `none`

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

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

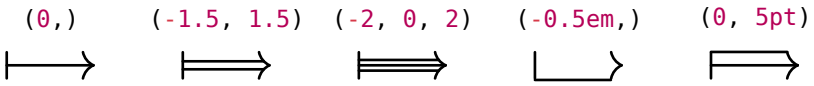
Apply a CeTZ path decoration to the stroke. Preset options are `"wave"`, `"zigzag"`, and `"coil"` (which may also be passed as convenience positional arguments), but a decoration function may also be specified.

```
#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)
  )
)
```



extrude `array` default `(0,)`

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.

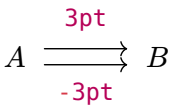


Notice how the ends of the line need to shift a little depending on the mark. This offset is computed with `cap-offset()`.

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

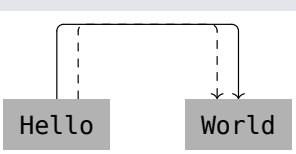
shift `length` or `number` or `pair` default `0pt`

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]),
)
```

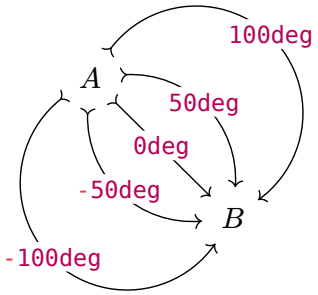


kind `string` default `auto`

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"`).

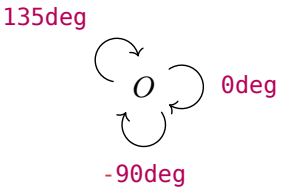
bend `angle` default `0deg`

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



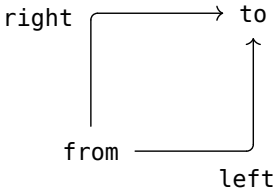
loop-angle `angle` default `none`

Angle around the node at which edge loops stick out at. Loops are arcs with the same start/end point and a large `bend` angle (e.g., `120deg`). This value has no effect for non-loop edges.



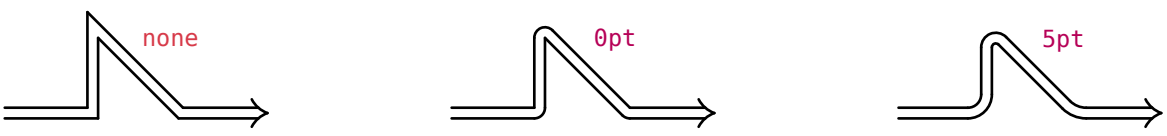
corner `none` or `left` or `right` default `none`

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



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!)

Default: the `edge-corner-radius` option of `diagram()`

`marks` `array` default `()`

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

$$M_i \in \texttt{fletcher.MARKS} = \left\{ \begin{array}{cccccc} \texttt{head}, & \texttt{doublehead}, & \texttt{triplehead}, & \texttt{harpoon}, & \texttt{straight}, & \texttt{solid}, \\ \texttt{stealth}, & \texttt{latex}, & \texttt{cone}, & \texttt{circle}, & \texttt{square}, & \texttt{diamond}, \\ \texttt{bar}, & \texttt{cross}, & \texttt{bracket}, & \texttt{parenthesis}, & \texttt{hook}, & \texttt{hooks}, \\ \texttt{>}, & \texttt{<}, & \texttt{>>}, & \texttt{<<}, & \texttt{>>>}, & \texttt{<<<}, \\ \texttt{|>}, & \texttt{<|}, & \texttt{}>}, & \texttt{<\{}, & \texttt{|}, & \texttt{||}, \\ \texttt{|||}, & \texttt{/}, & \texttt{\}, & \texttt{x}, & \texttt{X}, & \texttt{o}, \\ \texttt{0}, & \texttt{*}, & \texttt{@}, & \texttt{[]}, & \texttt{<>}, & \texttt{]}, \\ \texttt{[}, & \texttt{)}, & \texttt{(}, & \texttt{crowfoot}, & \texttt{n}, & \texttt{n!}, \\ \texttt{n?}, & \texttt{1}, & \texttt{1!}, & \texttt{1?}, & & \end{array} \right\}$$

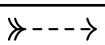
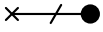
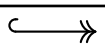
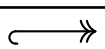
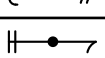
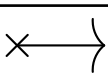
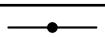
  is a mark name and

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

  is the line style.

- An array of mark names as strings or *mark objects* (dictionaries of parameters with a `draw` entry).

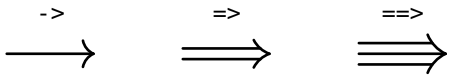
Shorthands are expanded into other arguments. For example, `edge(p1, p2, "=>")` is short for `edge(p1, p2, marks: (none, "head"), "double")`, or more precisely, the result of `edge(p1, p2, ..fletcher.interpret-marks-arg("=>"))`.

Result	Value of `marks`
	`"->"`
	`">>--->"`
	`"<=>"`
	`"==>"`
	`"->>-"`
	`"x-/-@"`
	`"|..|"`
	`"hook->>"`
	`"hook'->>"`
	`"||-*-harpoon'"`
	`("X", (inherit: "head", size: 15, sharpness: 40deg))`
	`((inherit: "circle", pos: 0.5, fill: auto),)`

`mark-scale` `percent` default `100%`

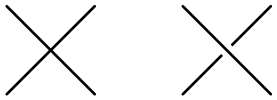
Scale factor for marks or arrowheads, relative to the `stroke` thickness. See also the `mark-scale` option of `diagram()`.

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



`crossing` `bool` default `false`

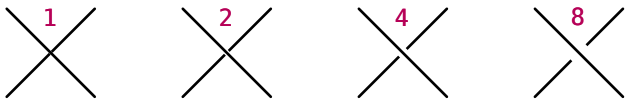
If `true`, draws a backdrop of color `crossing-fill` to give the illusion of lines crossing each other.



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

`crossing-thickness` `number`

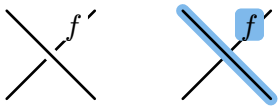
Thickness of the "crossing" background stroke (applicable if `crossing` is `true`) in multiples of the normal stroke's thickness.



Default: the `crossing-thickness` option of `diagram()`

`crossing-fill` `paint`

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



Default: the `crossing-fill` option of `diagram()`

`snap-to` `pair` default `(auto, auto)`

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

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.)

`layer` `number` default `0`

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()`.

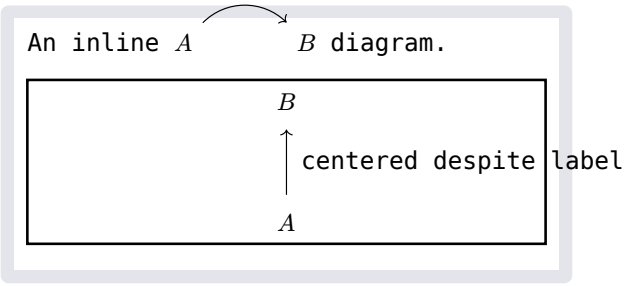
`floating` `bool` default `false`

Whether the edge should be *floating* so as not to affect the diagram's bounding box.

When `floating: true`, the edge is wrapped in `cetz.draw.floating(..)` which prevents the objects from affecting the canvas' bounding box.

```
 An inline #diagram($
  A edge(->, bend: #45deg, floating: #true) & B
$) diagram.

#rect(width: 7cm, align(center, diagram(
  node((0,1), $A$),
  edge("->", floating: true, [centered despite
label]),
  node((0,0), $B$),
)))
```



`post` `function` default `x => x`

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.

`..args` `any`

An edge's positional arguments may specify:

- the edge's `vertices`, each specified with a CeTZ-style coordinate
- the `label` content
- arrow `marks`, like `"=>"` or `"<<-|-o"`
- other style flags, like `"double"` or `"wave"`

Vertex coordinates must come first, and are optional:

```
edge(from, to, ..) // explicit start and end nodes
edge(to, ..) == edge(auto, to, ..) // start snaps to previous node
edge(..) == edge(auto, auto, ..) // snaps to previous and next nodes
edge(from, v1, v2, ..vs, to, ..) // a multi-segmented edge
edge(from, "->", to) // for two vertices, the marks style can come in between
```

All vertices except the start point can be shorthand relative coordinate string containing the characters {`l`, `r`, `u`, `d`, `t`, `b`, `n`, `e`, `s`, `w`} or commas.

If given as positional arguments, an edge's `marks` and `label` 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)
```

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`, `bracket`, `parenthesis`, `hook`, `hooks`, `>`, `<`, `>>`, `<<`, `>>>`, `<<<`, `|>`, `<|`, `}>`, `<{`, `|`, `||`, `|||`, `/`, `\`, `x`, `X`, `o`, `O`, `*`, `@`, `[]`, `<>`, `]`, `[`, `)`, `(`, `crowfoot`, `n`, `n!`, `n?`, `1`, `1!`, and `1?`.

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

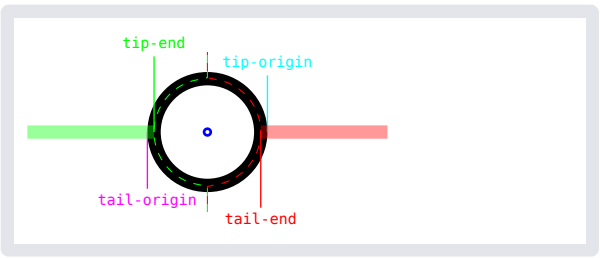
`cap-offset()`

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` default `1pt`

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

origin `point` default `(0,0)`

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

angle `angle` default `0deg`

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

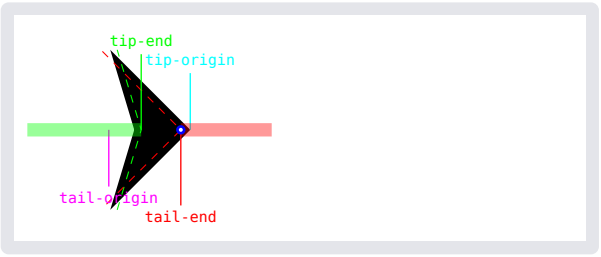
debug `bool` default `false`

Whether to draw the origin points.

`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 $\Longrightarrow$. 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` default `5pt`

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

`show-labels` `bool` default `true`

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

`show-offsets` `bool` default `true`

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

offset-range `number` default `6`

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

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)`. Shapes respect the `stroke`, `fill`, `width`, `height`, and `extrude` options of `edge()`.

- `rect()`
- `circle()`
- `ellipse()`
- `pill()`
- `parallelogram()`
- `trapezium()`
- `diamond()`
- `triangle()`
- `house()`
- `chevron()`
- `hexagon()`
- `octagon()`
- `stretched-glyph()`
- `cylinder()`
- `database()`

rect()

The standard rectangle node shape.

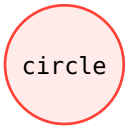
A string `"rect"` or the element function `rect` given to the `shape` option of `node()` are interpreted as this shape.


```
rect(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` default `1`

Scale factor for ellipse radii.

pill()

A capsule node shape.


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

parallelogram()

A slanted rectangle node shape.


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

angle `angle` default `20deg`

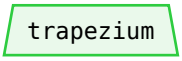
Angle of the slant, `0deg` is a rectangle. Don't set to `90deg` unless you want your document to be larger than the solar system.

fit `number` default `0.8`

Adjusts how comfortably the parallelogram fits the label's bounding box.

`trapezium()`

An isosceles trapezium node shape.




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

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

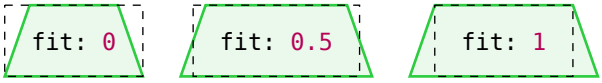
The side the shorter parallel edge is on.

angle `angle` default `20deg`

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.

fit `number` default `0.8`

Adjusts how comfortably the trapezium fits the label's bounding box.



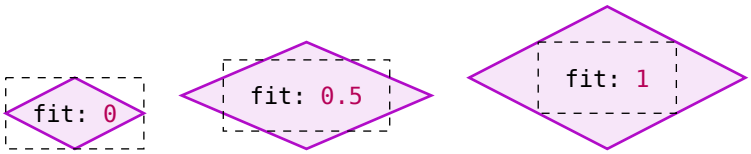
diamond()

A rhombus node shape.


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

fit `number` default `0.5`

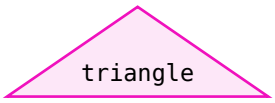
Adjusts how comfortably the diamond fits the label's bounding box.



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` default `top`

Direction the triangle points.

angle `angle` or `auto` default `auto`

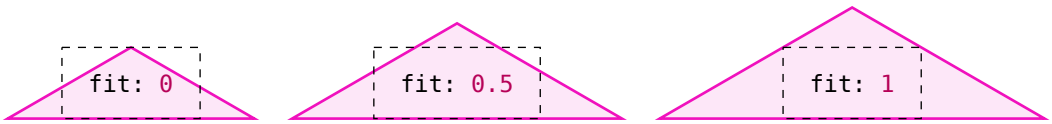
Angle of the triangle opposite the base.

aspect `number` or `auto` default `auto`

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

fit `number` default `0.8`

Adjusts how comfortably the triangle fits the label's bounding box.



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` default `top`

Direction of the roof of the house.

angle `angle` default `10deg`

The slant of the roof. A plain rectangle is `0deg`, and `90deg` is a sky scraper stretching past Pluto.

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` default `right`

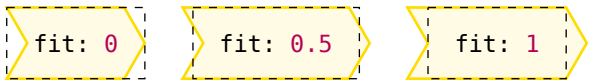
Direction the chevron points.

angle `angle` default `30deg`

The slant of the arrow. A plain rectangle is `0deg`.

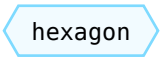
fit `number` default `0.8`

Adjusts how comfortably the chevron fits the label's bounding box.



`hexagon()`

An (irregular) hexagon node shape.




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

angle `angle` default `30deg`

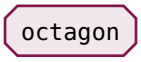
Half the exterior angle, `0deg` being a rectangle.

fit `number` default `0.8`

Adjusts how comfortably the hexagon fits the label's bounding box.

`octagon()`

A truncated rectangle node shape.




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

truncate `number` or `length` default `0.5`

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

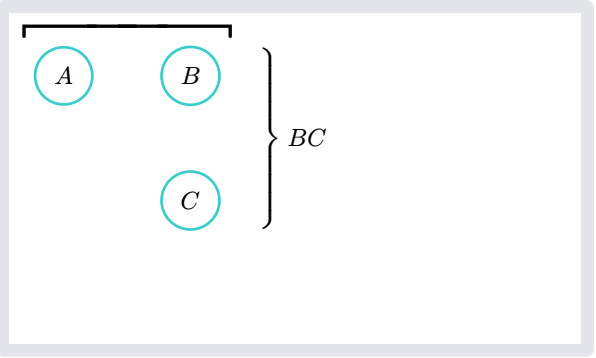
`stretched-glyph()`

A stretched glyph along one side of a node. See also `shapes.brace`, `shapes.bracket`, and `shapes.paren`, which are implemented using this shape.

Like this!

This is especially useful when used with `enclose` nodes.

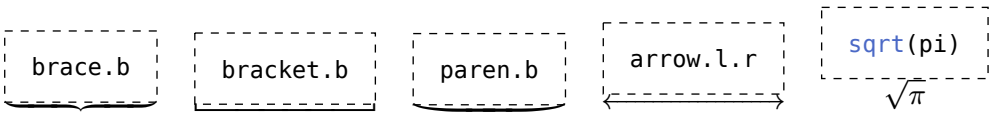
```
#import fletcher.shapes: brace, bracket
#diagram(
  spacing: 1cm,
  node-stroke: teal,
  node((0,0), $A$, name: <A>),
  node((1,0), $B$, name: <B>),
  node((1,1), $C$, name: <C>),
  node(enclose: (<A>, <B>), shape: bracket.with(
    dir: top, size: 2em)),
  node(enclose: (<B>, <C>), shape: brace.with(
    dir: right, length: 100% - 1em,
    sep: 10pt, label: $B C$)),
)
```




```
stretched-glyph(
  node,
  extrude,
  glyph: symbol content,
  dir: direction,
  sep: length,
  length: relative,
  label: content,
  label-sep: length,
  ..args: any,
)
```

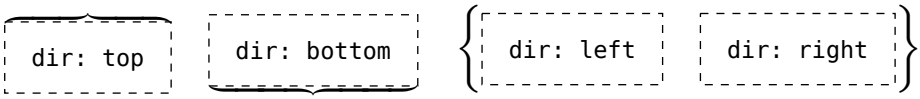
`glyph` `symbol` or `content` default `sym.brace.b`

The glyph to use. This works best with glyphs that can be stretched with the `stretch()` function, but any glyph or equation can be used.



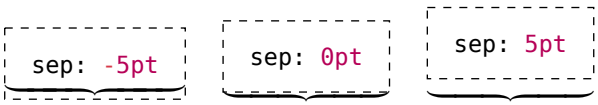
`dir` `direction` default `bottom`

The side of the node to place the glyph across. Note that the glyph must be chosen to match the direction.



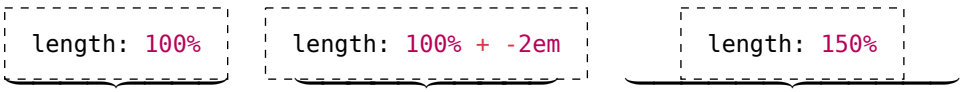
sep `length` default `0pt`

Extra distance between the glyph and the node's edge.



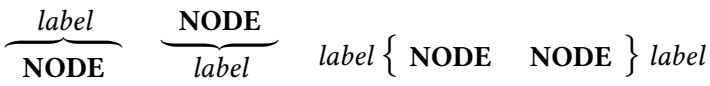
length `relative` default `100%`

Size of the glyph. A relative length such as `100% + 5pt` means `5pt` more than the size of the node. This is ultimately given to the `stretch()` function.



label `content` default `none`

Content to be placed at the top/bottom/left/right of the glyph, depending on `dir`.

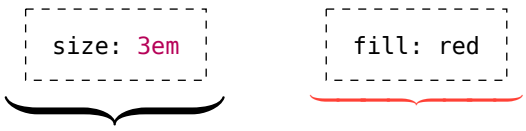


label-sep `length` default `0.25em`

Separation between label and glyph.

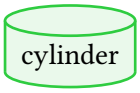
..args `any`

Arguments given to the `text()` element containing the glyph. Useful for changing color or the font size (defining overall scale without affecting its stretch length).



`cylinder()`

A pseudo-3D cylindrical shape.




```
cylinder(node, extrude)
```

database()

A database shape (stacked pseudo-3D cylindrical shape).


```
database(node, extrude)
```

coords.typ

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 rooted at a xy coordinate 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()`

Resolve CeTZ-style coordinate expressions to absolute vectors.

This is an drop-in replacement of `cetz.coordinate.resolve()` but extended to handle fletcher's elastic uv coordinates alongside CeTZ' physical xy coordinates. The target coordinate system must be specified in the context object `ctx`.

Resolving uv coordinates to or from xy coordinates requires the diagram's `grid`, which defines the non-linear maps `uv-to-xy()` and `xy-to-uv()`. The `grid` may be supplied in the context object `ctx`.

If `grid` is not supplied, **coordinate resolution may fail**, in which case the vector `(float.nan, float.nan)` is returned.

```
resolve(
  ctx: dictionary,
  update,
  ..coordinates: coordinate,
)
```

`ctx` `dictionary`

CeTZ canvas context object, additionally containing:
- `target-system`: the target coordinate system to resolve to, one of `"uv"` or `"xyz"`.
- `grid` (optional): the diagram's grid specification, defining the coordinate maps $uv \leftrightarrow xy$. If not given, coordinates requiring this map resolve to `(float.nan, float.nan)`.

`..coordinates` `coordinate`

CeTZ-style coordinate expression(s), e.g., `(1, 2)`, `(45deg, 2cm)`, or `(rel: (+1, 0), to: "name")`.

`diagram.typ`

- `interpret-axes()`
- `expand-fractional-rects()`
- `compute-cell-sizes()`
- `compute-cell-centers()`
- `compute-grid()`

`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 $\downarrow$.

`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.

This is the algorithm used to determine grid layout in diagrams.

```
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.

`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(
  flip: dictionary,
  verts: array,
  rects: array,
)
```

`flip` `dictionary`

Describes axis order and orientation.

`verts` `array`

Points that should be contained in the resulting grid.

`rects` `array`

Rectangles (dictionaries of the form `(center, size)` which are used to determine cell sizes.

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(
  rects,
  verts,
  options,
)
```

node.typ

- measure-node-size()
- resolve-node-enclosures()
- resolve-node-coordinates()

measure-node-size()

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.

```
measure-node-size(node)
```

resolve-node-enclosures()

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

```
resolve-node-enclosures(nodes, ctx)
```

`resolve-node-coordinates()`

Resolve node positions to a target coordinate system in sequence.

CeTZ-style coordinate expressions work, with the previous coordinate `()` referring to the resolved position of the previous node.

The resolved coordinates are added to each node's `pos` dictionary.

`resolve-node-coordinates(nodes: array, ctx: dictionary) -> array`

`nodes` `array`

Array of nodes, each a dictionary containing a `pos` entry, which should be a CeTZ-compatible coordinate expression.

`ctx` `dictionary` default `(:)`

CeTZ-style context to be passed to `resolve(ctx, ..)`. This must contain `target-system`, and optionally `grid`.

`edge.typ`

- `interpret-marks-arg()`
- `interpret-edge-args()`
- `apply-edge-shift()`

`interpret-marks-arg()`

Parse and interpret the marks argument provided to `edge()`. Returns a dictionary of processed `edge()` arguments.

`interpret-marks-arg(arg: string array) -> dictiony`

`arg` `string` or `array`

Can be a string, (e.g. `"->"`, `"<=>"`), etc, or an array of marks. A mark can be a string (e.g., `">"` or `"head"`, `"x"` or `"cross"`) or a dictionary containing the keys:

- `kind` (required) the mark name, e.g. `"solid"` or `"bar"`
- `pos` the position along the edge to place the mark, from 0 to 1
- `rev` whether to reverse the direction
- parameters specific to the kind of mark, e.g., `size` or `sharpness`

interpret-edge-args()

Interpret the positional arguments given to an edge()

Tries to intelligently distinguish the from, to, marks, and label arguments based on the argument types.

Generally, the following combinations are allowed:

```
edge(..<coords>, ..<marklabel>, ..<options>)
<coords> = () or (to) or (from, to) or (from, ..vertices, to)
<marklabel> = (marks, label) or (label, marks) or (marks) or (label) or ()
<options> = any number of options specified as strings
```

```
interpret-edge-args(args, 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

- place-edge-label-on-curve()
- draw-edge-line()
- draw-edge-arc()
- draw-edge-polyline()
- find-farthest-intersection()
- get-node-anchor()
- defocus-adjustment()
- draw-debug-axes()
- hide()

place-edge-label-on-curve()

Draw an edge label at point along a curve.

Label is drawn near the point curve(edge.label-pos), respecting the label options of edge() such as label-side and label-angle.

```
place-edge-label-on-curve(
  edge: dictionary,
  curve: function,
  debug,
)
```

edge `dictionary`

Edge object. Must include:
- `label-pos`
- `label-sep`
- `label-side`
- `label-anchor`
- `label-angle`
- `label-wrapper`

curve `function`

Parametric curve $\mathbb{R} \to \mathbb{R}^2$ describing the shape of the edge in xy coordinates.

`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` default `0`

Level of debug details to draw.

`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` default `0`

Level of debug details to draw.

`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` default `0`

Level of debug details to draw.

`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,
  floating,
)
```

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` default `true`

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

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` default `0pt`

Gradient for linear extrapolation beyond array bounds.

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 default 0pt

Gradient for linear extrapolation beyond array bounds.

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)
```