$$\begin{tikzcd} A \arrow[r, "f", tail] & B \end{tikzcd}$$

fletcher

(noun) a maker of arrows

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

Commutative diagrams, finite state machines, control systems block diagrams...

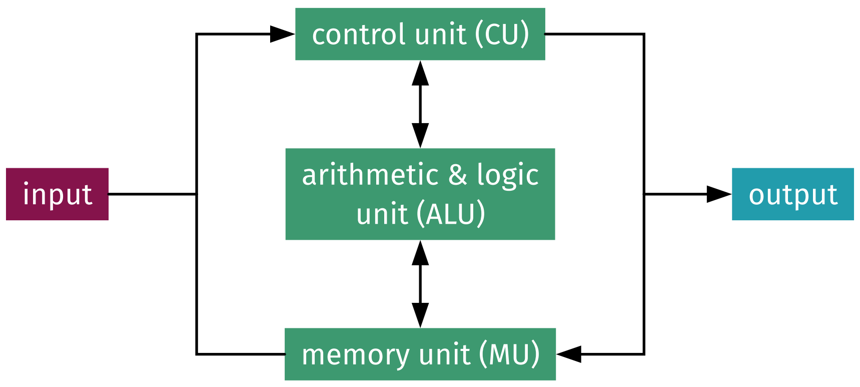
`github.com/Jollywatt/typst-fletcher`

Version 0.3.0 **(not yet stable)**

Contents

$$\begin{tikzcd}
 & TP \otimes \pi^* T^* X \arrow[rr, "d\lambda_g \otimes (\lambda_g \times 1)"] \arrow[dd, dotted, "c_\pi"'] & & TP \otimes \pi^* T^* X \arrow[rr, "\pi^G"] \arrow[dd, "c_\pi"'] \arrow[rr, <->, dotted, bend left=40, "\Omega"] & & T_G P \otimes T^* X \arrow[dd, "\bar{c}_\pi"'] \\
JP \arrow[ur, hook, "i_J"] \arrow[rr, "j\lambda_g"] \arrow[dd, Rrightarrow, "\pi_J"'] & & JP \arrow[ur, hook, "i_J"] \arrow[dd, "\pi_J"'] & & \mathrm{CP} \arrow[ur, hook, "i_C"] \arrow[dd, "\pi_{\mathrm{CP}}"'] & \\
 & \pi^*(TX \otimes T^* X) \arrow[rr, dotted, "\lambda_g \times 1"] & & \pi^*(TX \otimes T^* X) \arrow[rr, dotted, "\pi^G"] & & TX \otimes T^* X \\
P \arrow[ur, "\mathrm{Id}"] \arrow[rr, "\lambda_g"] & & P \arrow[ur, "\mathrm{Id}"] \arrow[rr, "\pi^G = \pi"] & & X \arrow[ur, "\mathrm{Id}"] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & \text{magma} \arrow[dll, "\text{Assoc}"'] \arrow[d, "\text{Id}"] \arrow[drr, "\text{Div}"] & & \\
\text{semigroup} \arrow[d, "\text{Id}"] \arrow[drr, "\text{Div}"] & & \text{unital magma} \arrow[dll, "\text{Assoc}"] \arrow[drr, "\text{Inv}"'] & & \text{quasigroup} \arrow[dll, "\text{Assoc}"] \arrow[d, "\text{Id}"] \\
\text{monoid} \arrow[drr, "\text{Inv}"] & & \text{inverse semigroup} \arrow[d, "\text{Id}"] & & \text{loop} \arrow[dll, "\text{Assoc}"'] \\
 & & \text{group} & &
\end{tikzcd}$$



Examples

```
#import "@preview/fletcher:0.3.0" as fletcher: node, edge
```

```
#fletcher.diagram({
  let (src, img, quo) = ((0, 1), (1, 1), (0, 0))
  node(src, $G$)
  node(img, $im f$)
  node(quo, $G slash ker(f)$)
  edge(src, img, $f$, "->")
  edge(quo, img, $tilde(f)$, "hook-->", label-side: right)
  edge(src, quo, $pi$, "->>")
})
```

$$\begin{tikzcd}
G \arrow[r, "f"] \arrow[d, two heads, "\pi"] & \operatorname{im} f \\
G/\ker(f) \arrow[ur, hook, dashed, "\tilde{f}"'] &
\end{tikzcd}$$

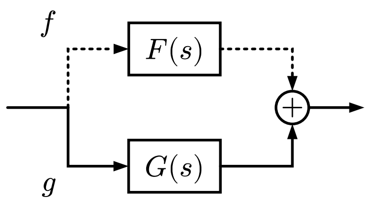
```
An equation $f: A -> B$ and \
a diagram #fletcher.diagram(
  node-inset: 4pt,
  node((0,0), $A$),
  edge((0,0), (1,0), text(0.8em, $f$), "->", label-sep: 1pt),
  node((1,0), $B$),
).
```

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

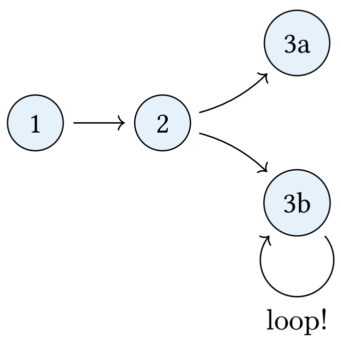
```
#fletcher.diagram(spacing: 2cm, {
  let (A, B) = ((0,0), (1,0))
  node(A, $cal(A)$)
  node(B, $cal(B)$)
  edge(A, B, $F$, "->", bend: +35deg)
  edge(A, B, $G$, "->", bend: -35deg)
  let h = 0.21
  edge((.5,+h), (.5,-h), $alpha$, "=>")
})
```

$$\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "F", ""{name=U, below}] \arrow[r, bend right, "G"', ""{name=D}] & \mathcal{B} \arrow[Rightarrow, from=U, to=D, "\alpha"']
\end{tikzcd}$$

```
#fletcher.diagram(
  spacing: (8mm, 3mm), // wide columns, narrow rows
  node-stroke: 1pt,    // outline node shapes
  edge-thickness: 1pt, // thickness of lines
  mark-scale: 60%,     // make arrowheads smaller
  edge((-2,0), (-1,0)),
  edge((-1,0), (0,+1), $f$, "..|>", corner: left),
  edge((-1,0), (0,-1), $g$, "-|>", corner: right),
  node((0,+1), $F(s)$),
  node((0,-1), $G(s)$),
  edge((0,+1), (1,0), "..|>", corner: left),
  edge((0,-1), (1,0), "-|>", corner: right),
  node((1,0), $ + $, inset: 1pt),
  edge((1,0), (2,0), "-|>"),
)
```



```
#fletcher.diagram(
  node-stroke: black + 0.5pt,
  node-fill: blue.lighten(90%),
  node-outset: 4pt,
  spacing: (15mm, 8mm),
  node((0,0), [1]),
  node((1,0), [2]),
  node((2,1), [3a]),
  node((2,-1), [3b]),
  edge((0,0), (1,0), "->"),
  edge((1,0), (2,+1), "->", bend: -15deg),
  edge((1,0), (2,-1), "->", bend: +15deg),
  edge((2,-1), (2,-1), "->", bend: +130deg, label: [loop!]),
)
```



Details

Nodes

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

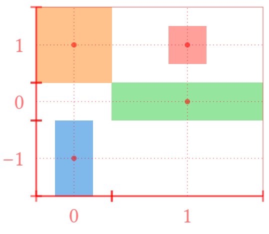
Nodes are content placed in the diagram at a particular coordinate. They fit to the size of their label (with an `inset` and `outset`), can be circular or rectangular (`shape`), and can be given a `stroke` and `fill`.

Elastic coordinates

Diagrams are laid out on a flexible coordinate grid. When a node is placed, the rows and columns grow to accommodate the node's size, like a table. See the `diagram()` parameters for more control: `node-size` is the minimum row and column width, and `spacing` is the gutter between rows and columns, respectively.

Elastic coordinates can be demonstrated more clearly with a debug grid and no spacing.

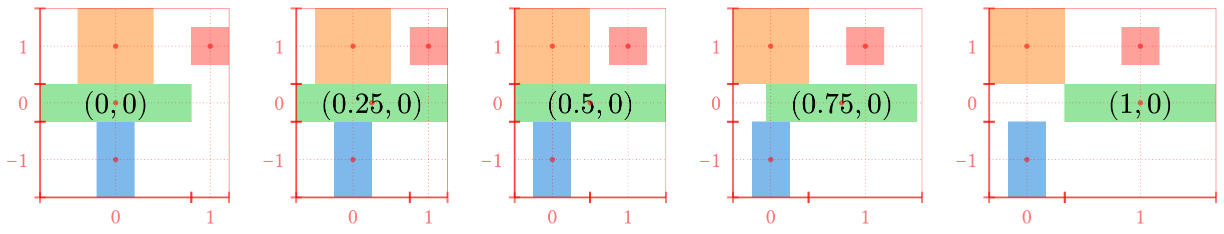
```
#let b(c, w, h) = box(fill: c.lighten(50%), width: w, height: h)
#fletcher.diagram(
  debug: 1,
  spacing: 0pt,
  node-inset: 0pt,
  node((0,-1), b(blue,    5mm, 10mm)),
  node((1, 0), b(green,  20mm,  5mm)),
  node((1, 1), b(red,     5mm,  5mm)),
  node((0, 1), b(orange, 10mm, 10mm)),
)
```



Fractional coordinates

Rows and columns are at integer coordinates, but nodes may have fractional coordinates. These are dealt with by linearly interpolating the diagram between what it would be if the coordinates were rounded up or down. Both the node's position and its influence on row/column sizes are interpolated.

As a result, diagrams are responsive to node sizes (like tables) while also allowing precise positioning.



Edges

`edge(node-1, node-2, label, marks, ..options)`

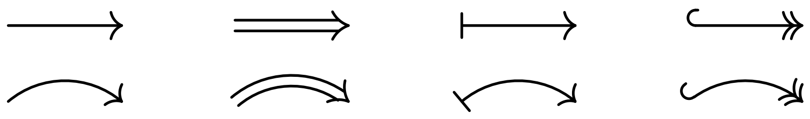
Edges connect two coordinates. If there is a node at an endpoint, the edge attaches to the nodes' bounding circle or rectangle. Edges can have `labels`, can `bend` into arcs, and can have various arrow `marks`.

```
#fletcher.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: -10deg, "dashed")
    edge(c, b, bend: +10deg, "dotted")
    edge(a, abc, $a$)
    edge(b, abc, "<=>")
    edge(c, abc, $c$)
})
```

$$\begin{tikzcd}
 & A \times B \times C & \\
A \arrow[ur, dash, "a"] \arrow[dr, dash, dashed, bend right] & & C \arrow[ul, dash, "c"'] \arrow[dl, dash, dotted, bend left] \\
 & B \arrow[uu, Leftrightarrow] &
\end{tikzcd}$$

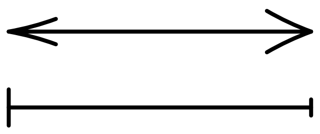
Marks and arrows

A few mathematical arrow heads are supported, designed to match the symbols $\to$, $\Rightarrow$, $\twoheadrightarrow$, $\hookrightarrow$, $\mapsto$, etc. See the `marks` argument of `edge()` for details.



Most marks have some parameters like size or sharpness angle that you can customize. This isn't a stable feature, but here's something to get you started:

```
#fletcher.diagram(
  edge-thickness: 1.5pt,
  spacing: (4cm, 1cm),
  {
    let custom-head = ( // sharper arrow head
      kind: "head",
      sharpness: 10deg,
      size: 70,
      delta: 10deg,
    )
    edge((0,1), (1,1), marks: (custom-head, custom-head + (sharpness: 20deg)))
    edge((0,0), (1,0), marks: ("bar", (kind: "bar", size: 2))) // smaller bar
  },
)
```

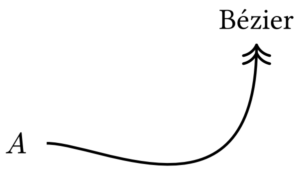


CeTZ integration

Currently, only straight, arc and right-angled connectors are supported. However, an escape hatch is provided with the `render` argument of `diagram()` so you can intercept diagram data and draw things using CeTZ directly.

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

```
#fletcher.diagram(
  node((0,0), $A$),
  node((2,1), [Bézier]),
  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, options)

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

      // get anchor points for the connector
      let p1 = fletcher.get-node-anchor(n1, 0deg)
      let p2 = fletcher.get-node-anchor(n2, -90deg)

      // make some control points
      let c1 = cetz.vector.add(p1, (20pt, 0pt))
      let c2 = cetz.vector.add(p2, (0pt, -70pt))

      cetz.draw.bezier(p1, p2, c1, c2)

      // place an arrow head at a given point and angle
      fletcher.draw-arrow-cap(p2, 90deg, 1pt + black, "twohead")
    })
  }
)
```



The `defocus` adjustment

For aesthetic reasons, lines connecting to a node need not focus to the node's exact center, especially if the node is short and wide or tall and narrow. Notice the difference the figures below. "Defocusing" the connecting lines can make the diagram look more comfortable.

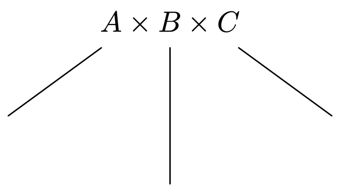


Figure 1: With defocus

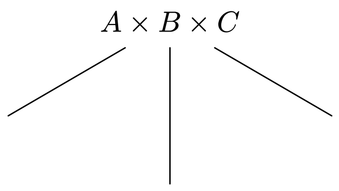


Figure 2: Without defocus

See the `node-defocus` argument of `diagram()` for details.

Function reference

diagram()

Draw an arrow diagram.

Parameters

```
diagram(
  ..objects: array ,
  debug: bool 1 2 3 ,
  spacing: length pair of lengths ,
  cell-size: length pair of lengths ,
  node-inset: length pair of lengths ,
  node-outset: length pair of lengths ,
  node-stroke: stroke ,
  node-fill: paint ,
  node-defocus: number ,
  label-sep,
  edge-thickness,
  mark-scale,
  crossing-fill: paint ,
  crossing-thickness: number ,
  render: function
)
```

..objects `array`

An array of dictionaries specifying the diagram's nodes and connections.

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

```
#fletcher.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), $D$)
  },
)
```

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

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

Default: `false`

spacing `length` or `pair of lengths`

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

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

Default: `3em`

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

Minimum size of all rows and columns.

Default: `0pt`

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

Default padding between a node's content and its bounding box.

Default: `12pt`

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

Default padding between a node's boundary and where edges terminate.

Default: `0pt`

node-stroke `stroke`

Default stroke for all nodes in diagram. Overridden by individual node options.

Default: `none`

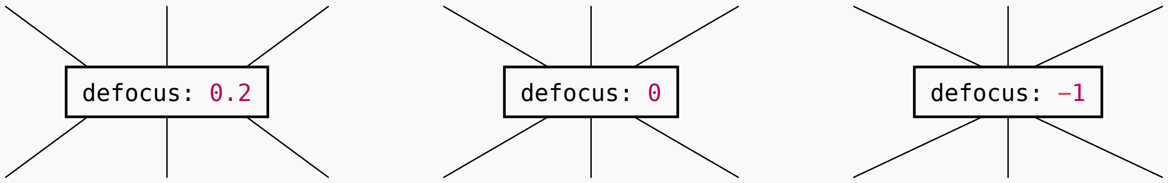
node-fill `paint`

Default fill for all nodes in diagram. Overridden by individual node options.

Default: `none`

node-defocus `number`

Default strength of the “defocus” adjustment for nodes. 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.



Default: `0.2`

label-sep

Default: `0.2em`

edge-thickness

Default: `0.048em`

mark-scale

Default: `100%`

crossing-fill `paint`

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

Default: `white`

crossing-thickness `number`

Default thickness of the occlusion made by crossing connectors. See the `crossing-thickness` option of `edge()`.

Default: `3`

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, options))
  }
```

`edge()`

Draw a connecting line or arc in an arrow diagram.

Parameters

```
edge(
  from: elastic coord ,
  to: elastic coord ,
  ..args: any ,
  label: content ,
  label-side: left  right  center ,
  label-pos: number ,
  label-sep: number ,
  label-anchor: anchor ,
  paint: paint ,
  thickness: length ,
  dash: dash type ,
  kind,
  bend: angle ,
  corner,
  marks: pair of strings ,
  mark-scale: percent ,
  extrude: array of numbers ,
  crossing: bool ,
  crossing-thickness: number ,
  crossing-fill: paint
)
```

from `elastic coord`

Start coordinate `(x, y)` of connector. If there is a node at that point, the connector is adjusted to begin at the node's bounding rectangle/circle.

to `elastic coord`

End coordinate `(x, y)` of connector. If there is a node at that point, the connector is adjusted to end at the node's bounding rectangle/circle.

..args `any`

The connector's `label` and `marks` named arguments can also be specified as positional arguments. For example, the following are equivalent:

```
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: "->")
```

label `content`

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

Default: `none`

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

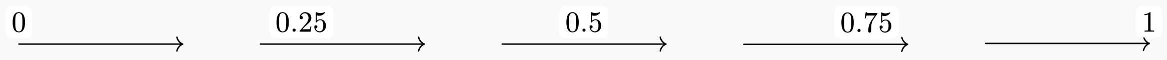
Which side of the connector to place the label on, viewed as you walk along it. If `center`, then the label is place over the connector. When `auto`, a value of `left` or `right` is chosen to automatically so that the label is

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

Default: `auto`

label-pos `number`

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

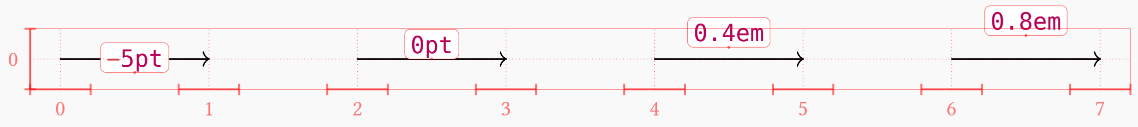


Default: `0.5`

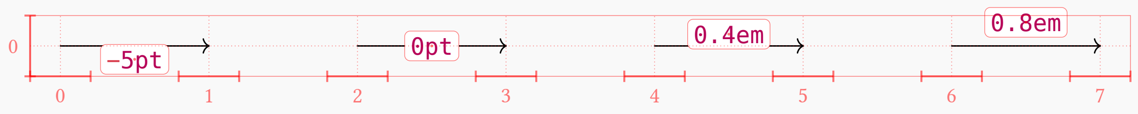
label-sep `number`

Separation between the connector and the label anchor.

With the default anchor (`"bottom"`):



With `label-anchor: "center"`:



Default: `auto`

label-anchor `anchor`

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

Default: `auto`

paint `paint`

Paint (color or gradient) of the connector stroke.

Default: `black`

thickness `length`

Thickness the connector stroke. Marks (arrow heads) scale with this thickness.

Default: `auto`

dash `dash type`

Dash style for the connector stroke.

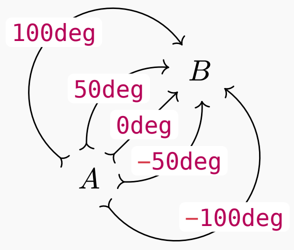
Default: `none`

kind

Default: `auto`

bend `angle`

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



Default: `0deg`

corner

Default: `none`

marks `pair of strings`

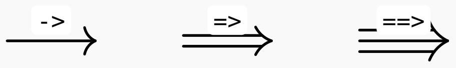
The start and end marks or arrow heads of the connector. A shorthand such as `"->"` can used instead. For example, `edge(p1, p2, "->")` is short for `edge(p1, p2, marks: (none, "head"))`.

Arrow	Shorthand	Arguments
——	`-`	`(marks: (none, none))`
- - - - - -	`--`	`(marks: (none, none), dash: "dashed")`
⋯⋯⋯	`..`	`(marks: (none, none), dash: "dotted")`
⟶	`->`	`(marks: (none, "head"))`
⟺	`<=>`	`( marks: ("doublehead", "doublehead"), extrude: (-2, 2), mark-scale: 110%, mark-variant: 2, )`
≻- - - -›	`>>-->`	`(marks: ("twotail", "head"), dash: "dashed")`
⊢⋯⋯⊣	`|..|`	`(marks: ("bar", "bar"), dash: "dotted")`
↪↠	`hook->>`	`(marks: ("hook", "twohead"))`
↪↠	`hook'->>`	`(marks: ("hook'", "twohead"))`
≻⇀	`>-harpoon`	`(marks: ("tail", "harpoon"))`
≻⇁	`>-harpoon'`	`(marks: ("tail", "harpoon'"))`

Default: `(none, none)`

mark-scale `percent`

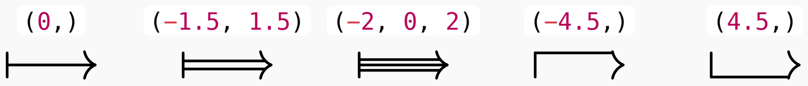
Scale factor for connector marks or arrow heads. This defaults to `100%` for single lines, `120%` for double lines and `150%` for triple lines. Does not affect the stroke thickness of the mark.



Default: `100%`

extrude `array of numbers`

Draw copies of the stroke extruded by the given multiple of the stroke thickness. Used to obtain doubling effect. Best explained by example:

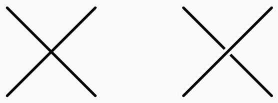


Notice how the ends of the line need to shift a little depending on the mark. For basic arrow heads, this offset is computed with `round-arrow-cap-offset()`.

Default: `(0,)`

crossing `bool`

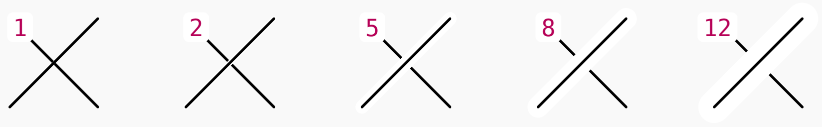
If `true`, draws a white backdrop to give the illusion of lines crossing each other.



Default: `false`

crossing-thickness `number`

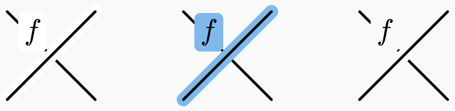
Thickness of the white "crossing" background stroke, if `crossing: true`, in multiples of the normal stroke's thickness.



Default: `auto`

crossing-fill `paint`

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



Default: `auto`

`node()`

Draw a labelled node in an arrow diagram.

Parameters

```
node(
  pos: point,
  label: content,
  inset: length auto,
  outset: length auto,
  shape: string auto,
  stroke: stroke,
  fill: paint,
  defocus: number
)
```

pos `point`

Dimensionless “elastic coordinates” `(x, y)` of the node, where `x` is the column and `y` is the row (increasing upwards). The coordinates are usually integers, but can be fractional.

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

label `content`

Node content to display.

inset `length` or `auto`

Padding between the node’s content and its bounding box or bounding circle. If `auto`, defaults to the `node-inset` option of `diagram()`.

Default: `auto`

outset `length` or `auto`

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

Default: `auto`

shape `string` or `auto`

Shape of the node, one of `"rect"` or `"circle"`. If `auto`, shape is automatically chosen depending on the aspect ratio of the node's label.

Default: `auto`

stroke `stroke`

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

Default: `auto`

fill `paint`

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

Default: `auto`

defocus `number`

Strength of the "defocus" adjustment for connectors incident with this node. If `auto`, defaults to the `node-defocus` option of `diagram()` .

Default: `auto`

`compute-grid()`

Determine the number, sizes and positions of rows and columns.

Parameters

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

nodes

options

`expand-fractional-rects()`

Convert an array of rects 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.

Parameters

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

rects `array of rects`

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

`get-edge-anchors()`

Get the points where a connector between two nodes should be drawn between, taking into account the nodes' sizes and relative positions.

Parameters

```
get-edge-anchors(
  edge: dictionary,
  nodes: pair of dictionaries
) -> pair of points
```

edge `dictionary`

The connector whose end points should be determined.

nodes `pair of dictionaries`

The start and end nodes of the connector.

`get-node-anchor()`

Get the point at which a connector should attach to a node from a given angle, taking into account the node's size and shape.

Parameters

```
get-node-anchor(
  node: dictionary,
  θ: angle
) -> point
```

node `dictionary`

The node to connect to.

θ `angle`

The desired angle from the node’s center to the connection point.

`interpret-mark()`

Take a string or dictionary specifying a mark and return a dictionary, adding defaults for any necessary missing parameters.

Parameters

```
interpret-mark(mark)
```

mark

`round-arrow-cap-offset()`

Calculate cap offset of round-style arrow cap

Parameters

```
round-arrow-cap-offset(
  r: length,
  θ: angle,
  y: length
)
```

r `length`

Radius of curvature of arrow cap.

θ `angle`

Angle made at the the arrow’s vertex, from the central stroke line to the arrow’s edge.

y `length`

Lateral offset from the central stroke line.

`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

Parameters

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

from `point`

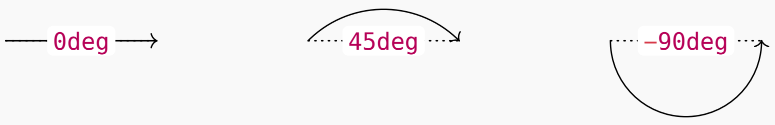
2D vector of initial point.

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.


0deg
45deg
-90deg