

$$A \rightrightarrows \overset{f}{\circ} \rightarrow B$$

fletcher

(noun) a maker of arrows

A [Typst](#) package for diagrams with lots of arrows, built on top of [CeTZ](#).

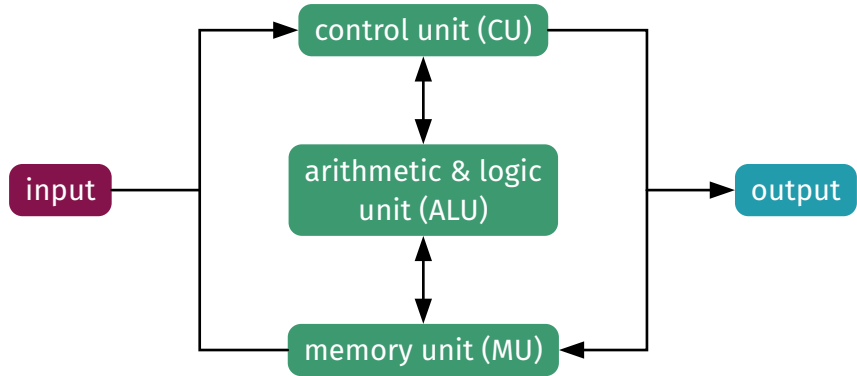
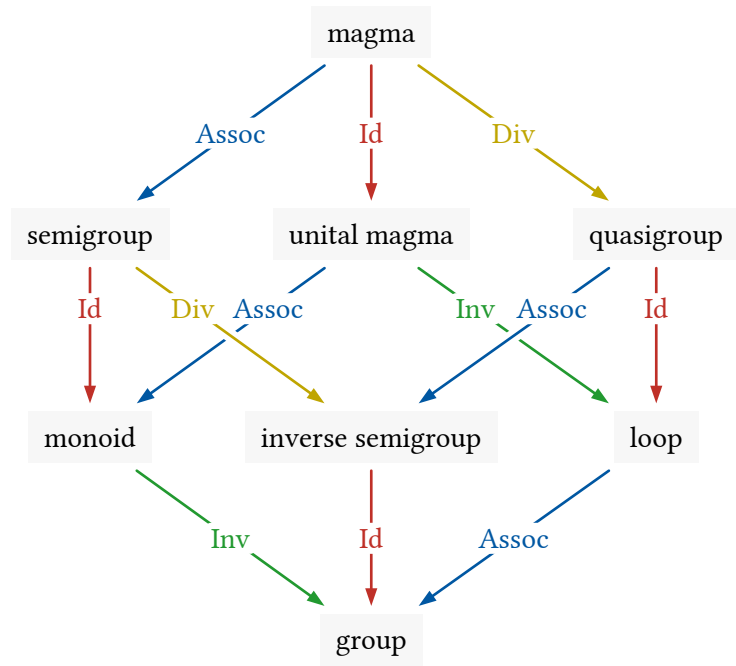
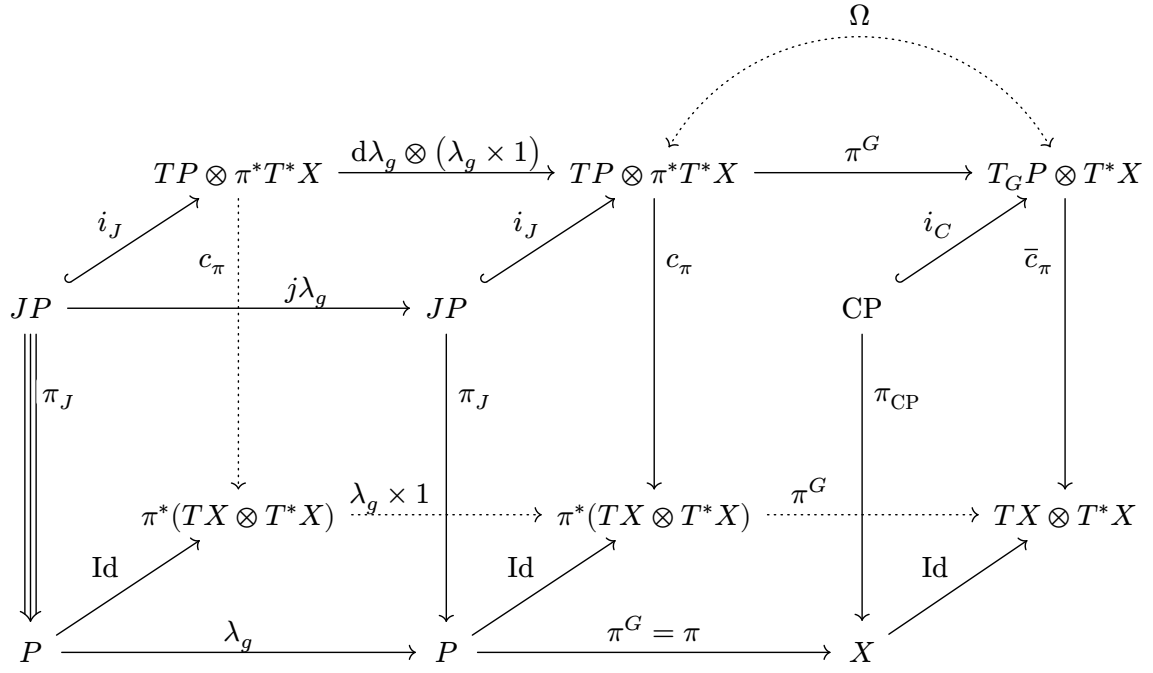
Commutative diagrams, finite state machines, control systems block diagrams...

github.com/Jollywatt/typst-fletcher

Version 0.4.0 (not yet stable)

Contents

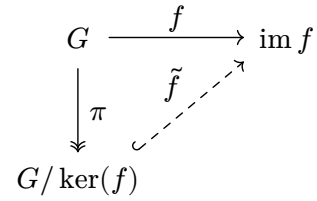
Getting started	3
Details	4
Nodes	4
Elastic coordinates	4
Fractional coordinates	4
Edges	5
Marks and arrows	5
Customised marks	6
Hanging tail correction	7
CeTZ integration	7
The defocus adjustment	7
Function reference	9
diagram	9
edge	12
node	17
compute-grid	20
expand-fractional-rects	20
get-edge-anchors	21
get-node-anchor	21
interpret-mark	21
interpret-marks-arg	22
round-arrow-cap-offset	22
get-arc-connecting-points	23



Getting started

```
#import "@preview/fletcher:0.4.0" as fletcher: node, edge
```

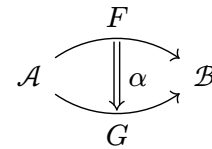
```
#fletcher.diagram({
  let (src, img, quo) = ((0, 1), (1, 1), (0, 0))
  node(src, $G$)
  node(img, $im f$)
  node(quo, $G \text{ slash } \ker(f)$)
  edge(src, img, $f$, "->")
  edge(quo, img, $\tilde{f}$, "hook-->", label-side: right)
  edge(src, quo, $\pi$, "->>")
})
```



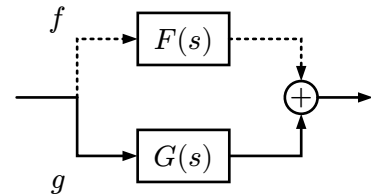
```
An equation $f: A \to B$ and \
a diagram #fletcher.diagram(
  node-inset: 4pt,
  node((0,0), $A$),
  edge((0,0), (1,0), text(0.8em, $f$), "->", label-sep: 1pt),
  node((1,0), $B$),
).
```

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

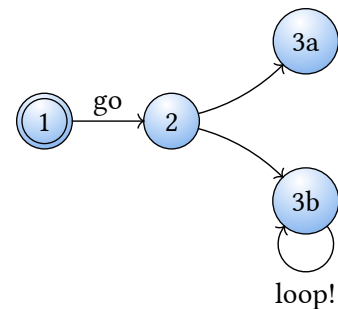
```
#fletcher.diagram(spacing: 2cm, {
  let (A, B) = ((0,0), (1,0))
  node(A, $cal(A)$)
  node(B, $cal(B)$)
  edge(A, B, $F$, "->", bend: +35deg)
  edge(A, B, $G$, "->", bend: -35deg)
  let h = 0.21
  edge((.5,+h), (.5,-h), $\alpha$, "=>")
})
```



```
#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: gradient.radial(white, blue, center: (40%, 20%),
    radius: 150%),
  spacing: (15mm, 8mm),
  node((0,0), [1], extrude: (0, -4)), // double stroke effect
  node((1,0), [2]),
  node((2,1), [3a]),
  node((2,-1), [3b]),
  edge((0,0), (1,0), [go], "->"),
  edge((1,0), (2,+1), "->", bend: -15deg),
  edge((1,0), (2,-1), "->", bend: +15deg),
  edge((2,-1), (2,-1), "->", bend: +130deg, label: [loop!]),
)
```



Details

The recommended way to load the package is:

```
#import "@preview/fletcher:0.4.0" as fletcher: node, edge
```

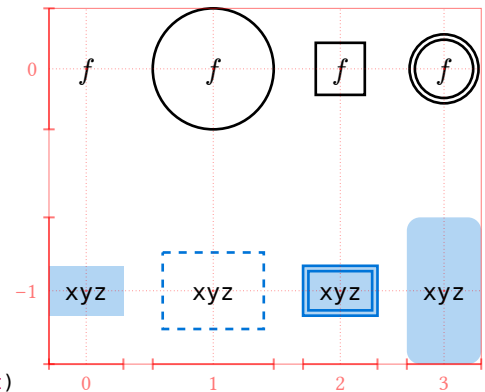
Other functions (including internal functions) are exported, so avoid importing everything with `*` and access them as needed with, e.g., `fletcher.diagram`.

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.

```
#fletcher.diagram(
  debug: 1,
  spacing: (1em, 3em), // (x, y)
  node((0, 0), $f$),
  node((1, 0), $f$, stroke: 1pt, radius: 8mm),
  node((2, 0), $f$, stroke: 1pt, shape: "rect"),
  node((3, 0), $f$, stroke: 1pt, extrude: (0, 2)),
  {
    let b = blue.lighten(70%)
    node((0,-1), `xyz`, fill: b, )
    let dash = (paint: blue, dash: "dashed")
    node((1,-1), `xyz`, stroke: dash, inset: 2em)
    node((2,-1), `xyz`, fill: b, stroke: blue, extrude: (0, -2))
    node((3,-1), `xyz`, fill: b, height: 5em, corner-radius: 5pt)
  }
)
```

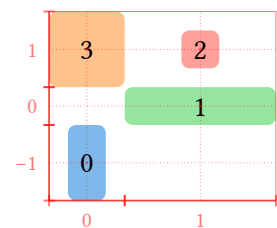


Elastic coordinates

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: `cell-size` is the minimum row and column width, and `spacing` is the gutter between rows and columns.

Elastic coordinates can be demonstrated more clearly with a debug grid and no spacing between cells:

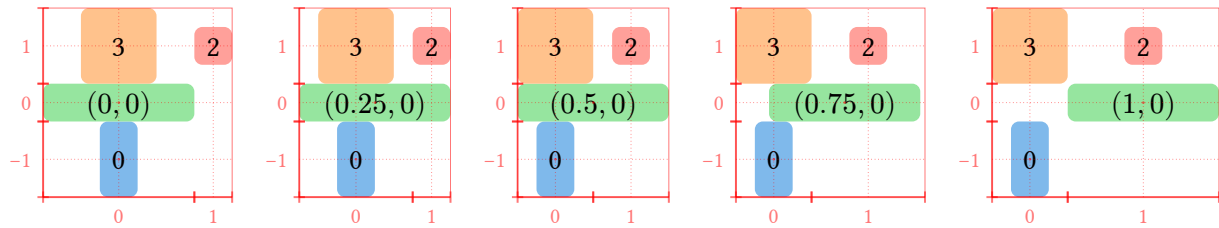
```
#let c = (blue, green, red, orange).map(x => x.lighten(50%))
#fletcher.diagram(
  debug: 1,
  spacing: 0pt,
  node-corner-radius: 3pt,
  node((0,-1), [0], fill: c.at(0), width: 5mm, height: 10mm),
  node((1, 0), [1], fill: c.at(1), width: 20mm, height: 5mm),
  node((1, 1), [2], fill: c.at(2), width: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

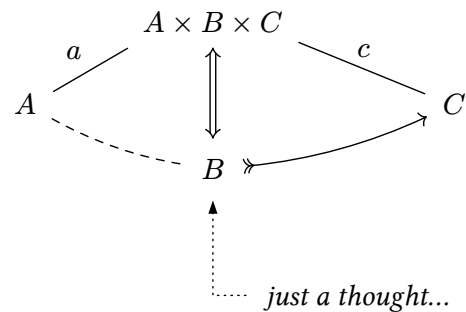
`edge(node-1, node-2, label, marks, ..options)`

Edges connect two coordinates. If there is a node at an endpoint, the edge attaches to the nodes' bounding shape. 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, $C$)

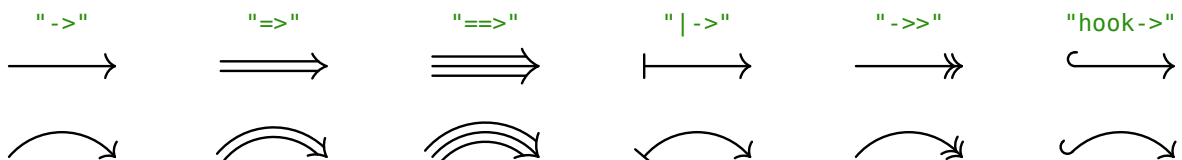
  edge(a, b, bend: -10deg, "dashed")
  edge(c, b, bend: +10deg, "<-<")
  edge(a, abc, $a$)
  edge(b, abc, "<=>")
  edge(c, abc, $c$)

  node((0.6, -3), [_just a thought..._])
  edge(b, (0.6, -3), "<|..", corner: right)
})
```

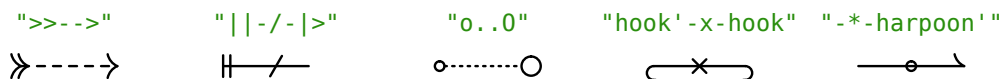


Marks and arrows

A few mathematical arrow heads are supported, designed to match \rightarrow , \Rightarrow , \implies , \mapsto , \rightsquigarrow , \hookrightarrow , etc.



Some other miscellaneous caps are supported, and marks can be placed anywhere along the edge.



Edge styles can be specified like `edge(a, b, "->")`, or by passing a dictionary of mark parameters:

```
#fletcher.diagram(
  edge-thickness: 2pt,
  spacing: 4cm,
  edge((0,0), (1,0), marks: (
    "x",
    (kind: "head", rev: true, pos: 0.5),
    (kind: "bar", size: 1, pos: 0.7),
    (kind: "solidhead", rev: true),
  ))
)
```



See the marks argument of `edge()` for details.

Customised marks

While convenient shorthands exist for specifying marks and stroke styles, finer control is possible. Shorthands like "`<->`" expand into specific `edge()` options. For example, `edge(a, b, "|=>")` is equivalent to `edge(a, b, marks: ("bar", "doublehead"), extrude: (-2, 2))`. Mark names such as "`bar`" or "`doublehead`" are in turn shorthand for a dictionary defining the mark's parameters. These parameters can be retrieved from the mark name as follows:

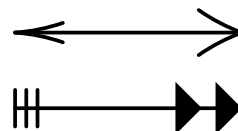
```
#fletcher.interpret-mark("doublehead")
// In this particular example:
// - `kind` selects the type of arrow head
// - `size` controls the radius of the arc
// - `sharpness` is (half) the angle of the tip
// - `delta` is the angle spanned by the arcs
// - `tail` is approximately the distance from the cap's tip to
//   the end of its arms. This is used to calculate a "tail hang"
//   correction to the arrowhead's bearing for tightly curved edges.
// Distances are multiples of the stroke thickness.
(
  kind: "head",
  size: 10.56,
  sharpness: 19deg,
  delta: 43.7deg,
  outer-len: 5.5,
)
```

Finally, the fully expanded version of a marks shorthand can be inspected by invoking `interpret-marks-arg()`:

```
#fletcher.interpret-marks-arg("|=>")
// `edge(..args, marks: "|=>")` is equivalent to
// `edge(..args, ..fletcher.interpret-marks-arg("|=>"))`
(
  marks: (
    (
      size: 4.9,
      angle: 0deg,
      pos: 0,
      rev: true,
      kind: "bar",
    ),
    (
      pos: 1,
      rev: false,
      kind: "head",
      size: 10.56,
      sharpness: 19deg,
      delta: 43.7deg,
      outer-len: 5.5,
    ),
  ),
  extrude: (-2, 2),
)
```

You can customise these basic marks by adjusting these parameters. For example:

```
#let my-head = (kind: "head", sharpness: 4deg, size: 50, delta: 15deg)
#let my-bar = (kind: "bar", extrude: (0, -3, -6))
#let my-solidhead = (kind: "solidhead", sharpness: 45deg)
#fletcher.diagram(
  edge-thickness: 1.4pt,
  spacing: (3cm, 1cm),
  edge((0,1), (1,1), marks: (my-head, my-head + (sharpness: 20deg))),
  edge((0,0), (1,0), marks: (my-bar, my-solidhead + (pos: 0.8), my-solidhead)),
)
```

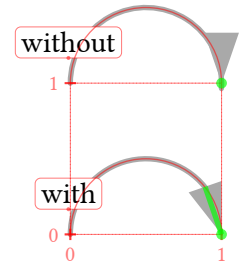


The particular marks and parameters are hard-wired and will likely change as this package is updated (so they are not documented). However, for finer control, you are encouraged to use the functions `interpret-marks-arg()` and `interpret-mark()` to discover the parameters.

Hang tail correction

All marks accept an `outer-len` parameter, the effect of which can be seen below:

```
#fletcher.diagram(  
  edge-thickness: 2pt,  
  spacing: 2cm,  
  debug: 4,  
  
  edge((0,1), (1,1), stroke: gray, bend: 90deg, label-pos: 0.1, label: [without],  
    marks: (none, (kind: "solidhead", outer-len: 0))),  
  edge((0,0), (1,0), stroke: gray, bend: 90deg, label-pos: 0.1, label: [with],  
    marks: (none, (kind: "solidhead"))), // use default hang  
)
```



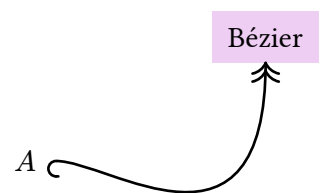
The tail length (specified in multiples of the stroke thickness) is the distance that the arrow head visually extends backwards over the stroke. This is visualised by the green line shown above. The mark is rotated so that the ends of the line both lie on the arc.

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], 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, 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, -80pt))  
  
      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")  
      fletcher.draw-arrow-cap(p1, 180deg, 1pt + black, (kind: "hook", outer-len: 0))  
    })  
  }  
)
```



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.

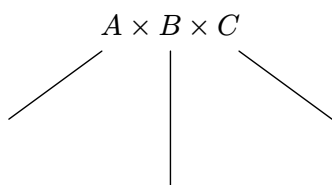


Figure 1: With defocus

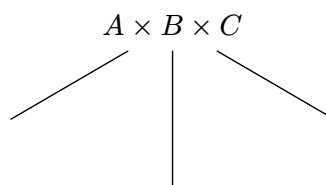


Figure 2: Without defocus

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-corner-radius ,  
  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.

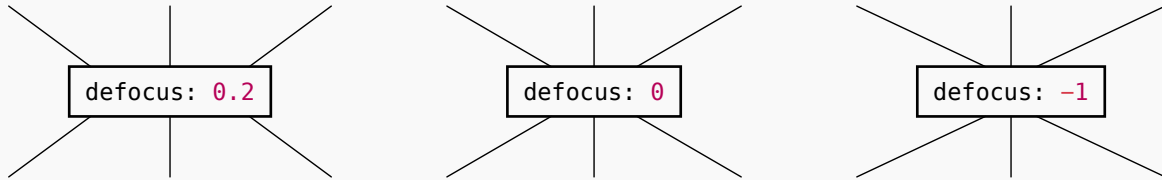
Default: **none**

node-corner-radius

Default: **0pt**

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: **5**

render **function**

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

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

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

```
Default: (grid, nodes, edges, options) => {  
  cetz.canvas(draw-diagram(grid, nodes, edges, 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 ,  
  stroke: stroke ,  
  dash: dash type ,  
  kind ,  
  bend: angle ,  
  corner ,  
  marks: pair of strings ,  
  mark-scale: percent ,  
  extrude: array ,  
  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), $$, "->")
edge((0,0), (1,0), $$, marks: "->")
edge((0,0), (1,0), "->", label: $$)
edge((0,0), (1,0), label: $$, 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 placed 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).

0 0.25 0.5 0.75 1

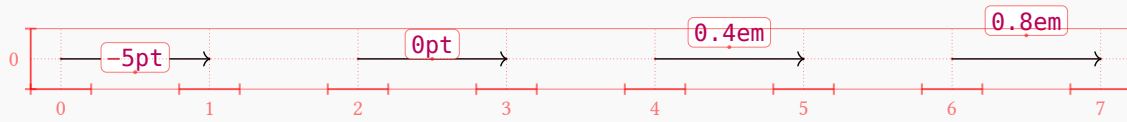


Default: `0.5`

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**

stroke stroke

Stroke style of the edge. Arrows scale with the stroke thickness.

Default: black

dash dash type

Dash style for the connector stroke.

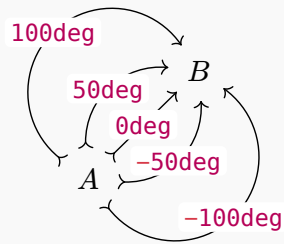
Default: **none**

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

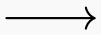
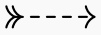
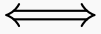
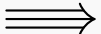
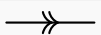
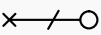
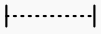
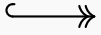
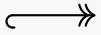
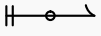
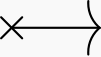
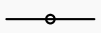
- A shorthand string such as `"->"` or `"hook' - / ->"`. Specifically, shorthand strings are of the form M_1LM_2 or $M_1LM_2LM_3$, where

$$M_i \in \{>, >>, <, <<, |>, <|, |, ||, /, \backslash, x, X, o, O, *, @\} \cup N$$

is a mark symbol and $L \in \{-, --, \dots, =, ==\}$ is the line style. The mark symbol can also be a name, $M_i \in N = \{\text{hook}, \text{hook}', \text{harpoon}, \text{harpoon}', \text{head}, \text{circle}, \dots\}$ where a trailing ' means to reflect the mark across the stroke.

- An array of marks, where each mark is specified by name or by a dictionary of parameters.

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

Arrow	marks
	<code>"->"</code>
	<code>">>-->"</code>
	<code>"<=>"</code>
	<code>"==>"</code>
	<code>"->>-"</code>
	<code>"x-/-@"</code>
	<code>" . . "</code>
	<code>"hook->"</code>
	<code>"hook' ->"</code>
	<code>" -* -harpoon'"</code>
	<code>("X", (kind: "head", size: 15, sharpness: 40deg))</code>
	<code>((kind: "circle", pos: 0.5, fill: true),)</code>

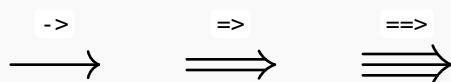
Default: `(none, none)`

mark-scale percent

Scale factor for marks or arrowheads.



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



Default: `100%`

extrude array

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



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 backdrop of color `crossing-fill` to give the illusion of lines crossing each other.



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

Default: **false**

crossing-thickness number

Thickness of the "crossing" background stroke, if `crossing: true`, in multiples of the normal stroke's thickness. Defaults to the `crossing-thickness` option of `diagram()`.



Default: **auto**

crossing-fill paint

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



Default: **auto**

node()

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

Parameters

```
node(  
  pos: point,  
  label: content,  
  inset: length auto,  
  outset: length auto,  
  shape: string auto,  
  width,  
  height,  
  radius,  
  stroke: stroke,  
  fill: paint,  
  corner-radius,  
  defocus: number,  
  extrude: array  
)
```

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.

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

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`

width

Default: **auto**

height

Default: **auto**

radius

Default: **auto**

stroke stroke

Stroke style for the node outline. 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**

corner-radius

Default: **auto**

defocus number

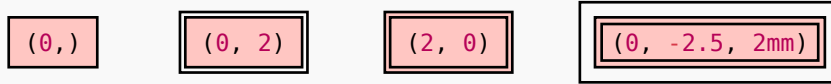
Strength of the “defocus” adjustment for connectors incident with this node. If **auto**, defaults to the `node-defocus` option of `diagram()`.

Default: **auto**

extrude `array`

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

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



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

Default: `(0,)`

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 fractional 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 to 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.

Ensures all required parameters except rev and pos are present.

Parameters

```
interpret-mark(  
    mark,  
    defaults  
)
```

mark

defaults

Default: (:)

interpret-marks-arg()

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

Parameters

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

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. "solidhead" or "bar"
- pos the position along the edge to place the mark, from 0 to 1
- rev whether to reverse the direction
- tail the visual length of the mark's tail
- parameters specific to the kind of mark, e.g., size or sharpness

round-arrow-cap-offset()

Calculate cap offset of round-style arrow cap, $r \left(\sin \theta - \sqrt{1 - \left(\cos \theta - \frac{|y|}{r} \right)^2} \right)$.

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.

