

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

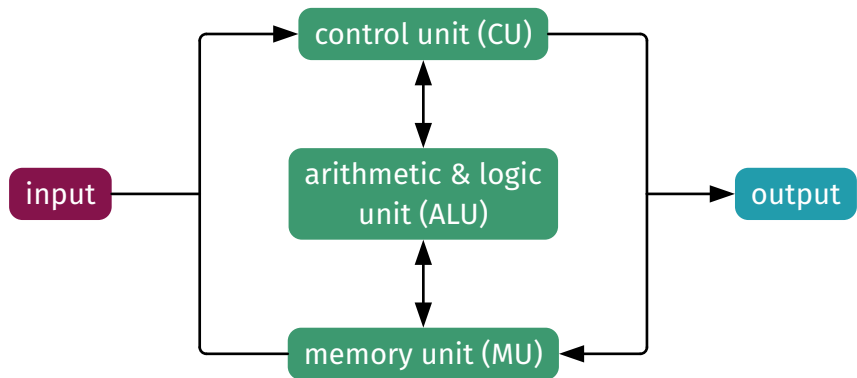
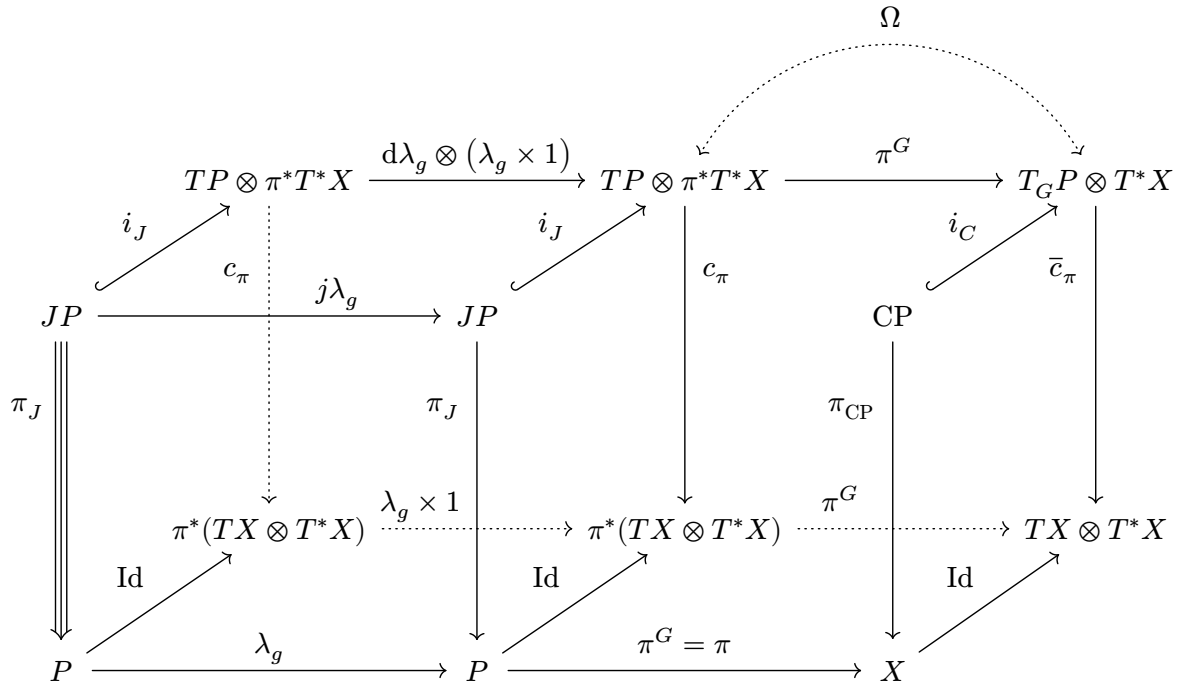
Commutative diagrams, finite state machines, block diagrams...

github.com/Jollywatt/typst-fletcher

Version 0.4.2

Contents

Usage examples	3
Nodes	4
Elastic coordinates	4
Fractional coordinates	4
Node shapes	5
Edges	5
Implicit coordinates	5
Relative coordinates	6
Edge types	6
The defocus adjustment	6
Marks and arrows	7
Adjusting marks	7
Hanging tail correction	8
CeTZ integration	9
Bézier edges	9
Node groups	9
Function reference	11
<code>diagram()</code>	11
<code>edge()</code>	14
<code>interpret-edge-args()</code>	21
<code>node()</code>	21
Marks	23
<code>interpret-mark()</code>	24
<code>interpret-marks-arg()</code>	24
<code>round-arrow-cap-offset()</code>	24
Behind the scenes	25
<code>get-arc-connecting-points()</code>	25
<code>lerp-at()</code>	26
<code>compute-grid()</code>	26
<code>expand-fractional-rects()</code>	27
<code>find-farthest-intersection()</code>	27



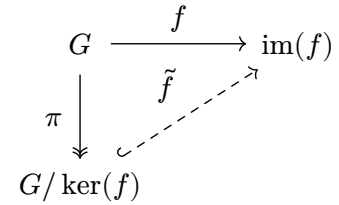
Usage examples

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

```
#import "@preview/fletcher:0.4.2" as fletcher: node, edge
```

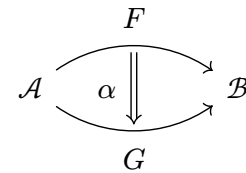
// You can specify nodes in math-mode, separated by ``&``:

```
#fletcher.diagram($
  G edge(f, ->) edge("d", pi, ->) & im(f) \
  G slash ker(f) edge("ur", tilde(f), "hook->")
$)
```

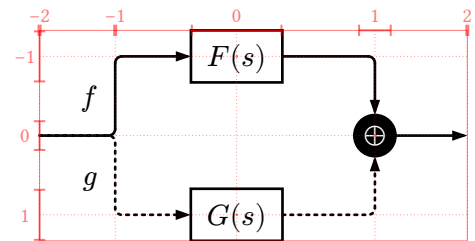


// Or you can use code-mode, with variables, loops, etc:

```
#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 = 0.2
  edge((.5,-h), (.5,+h), $alpha$, "=>")
})
```



```
#fletcher.diagram(
  debug: true, // show a coordinate grid
  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), $G(s)$),
  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("-|>"),
)
```



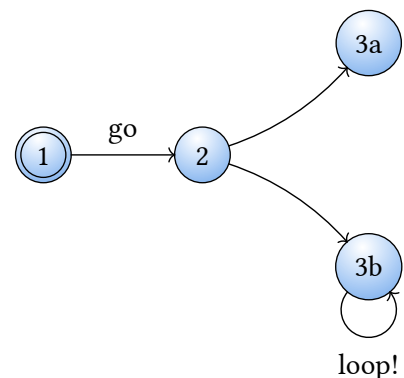
An equation `$f: A -> B$` and `\`

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

An equation $f : A \rightarrow B$ and

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

```
#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!]),
)
```



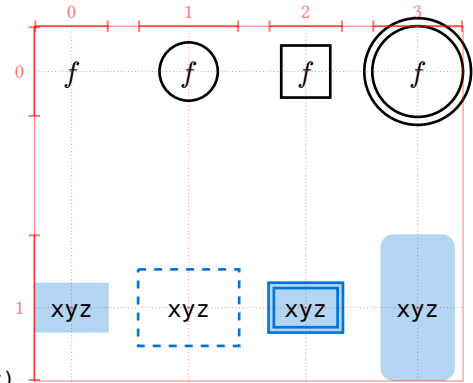
Nodes

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

Nodes are content centered at a particular coordinate. They automatically fit to the size of their label (with an inset and outset). They can be given a stroke and fill and be of any shape.

By default, the coordinates (x, y) are x going \rightarrow and y going \downarrow . This can be changed with the `axis` option of `diagram()`.

```
#fletcher.diagram(
  debug: 1,
  spacing: (1em, 4em), // (x, y)
  node((0,0), $f$,
  node((1,0), $f$, stroke: 1pt),
  node((2,0), $f$, stroke: 1pt, 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)
  }
)
```

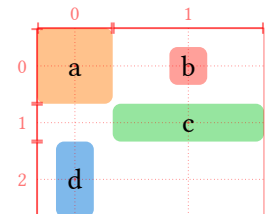


Elastic coordinates

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 = (orange, red, green, blue).map(x => x.lighten(50%))
#fletcher.diagram(
  debug: 1,
  spacing: 0pt,
  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),
)
```

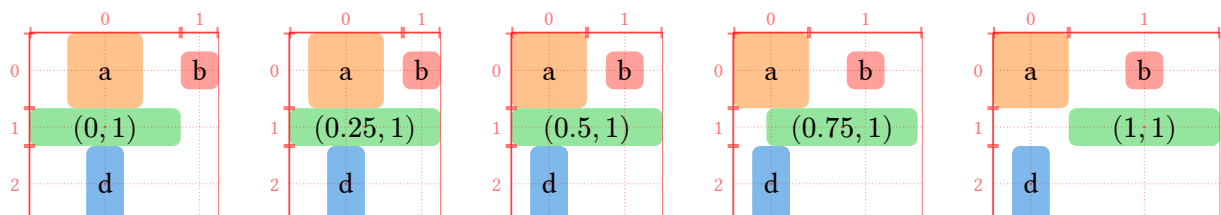


So far, this is just like a table. However, coordinates can also be fractional.

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.

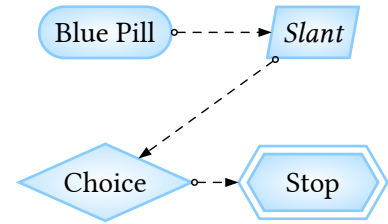
For example, see how the column sizes change as the green box moves from $(0, 0)$ to $(1, 0)$:



Node shapes

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

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



Custom shapes are also supported, but it is up to the user implement outline extrusion; see the `shape` option of `node()` for details.

Edges

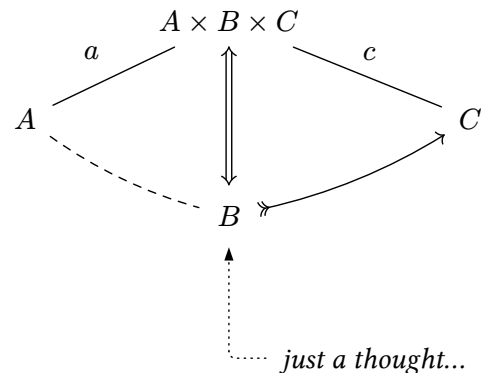
`edge(from, to, label, marks, ..options)`

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

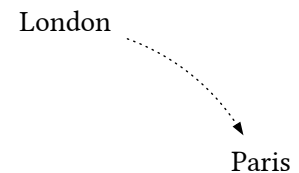
  node((.6,3), [_just a thought..._])
  edge(b, "..|>", corner: right)
})
)
```



Implicit coordinates

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

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



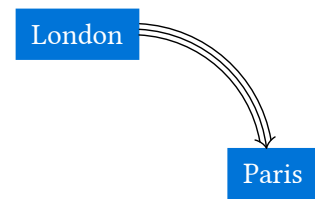
Implicit coordinates can be handy for diagrams in math-mode:

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



However, don't forget you can also use variables in code-mode to avoid repeating coordinates:

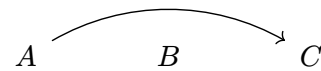
```
#fletcher.diagram(node-fill: blue, {
  let (dep, arv) = ((0,0), (1,1))
  node(dep, text(white)[London])
  node(arv, text(white)[Paris])
  edge(dep, arv, "==>", bend: 40deg)
})
```



Relative coordinates

It can also be handy to specify the direction of an edge, instead of its end coordinate. This can be done with `edge((x, y), (rel: (Δx, Δy)))`. For convenience, you can also specify a relative coordinate with string of *directions*, e.g., "u" for up or "br" for bottom right. Any combination of **top/up/north**, **bottomp/down/south**, **left/west**, and **right/east** are allowed. Together with implicit coordinates, this allows you do to things like:

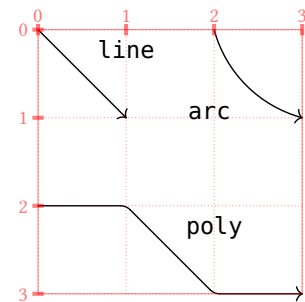
```
#fletcher.diagram($ A edge("rr", ->, bend: #30deg) & B & C $)
```



Edge types

Currently, there are three different kinds of edges: "line", "arc", and "poly". All nodes have a start and end point (from and to), and "poly" edges can also have an array of additional vertices. The kind defaults to "arc" if a bend is specified, and to "poly" if any vertices are given.

```
#fletcher.diagram(
  debug: 1,
  edge((0,0), (1,1), "->", `line`),
  edge((2,0), (3,1), "->", bend: -30deg, `arc`),
  edge((0,2), (3,3), vertices: ((1,2), (2,3)), "->", `poly`),
)
```



An alternative way to specify vertices is by providing multiple coordinates: `edge(A, B, C, D)` is the same as `edge(from: A, to: D, vertices: (B, C))` if the arguments are all coordinates. An edge's vertices and to coordinates can be relative (see above), so that the "poly" edge above could also be written in these ways:

```
edge((0,2), (rel: (1,0)), (rel: (1,1)), (rel: (1,0)), "->", `poly`)
edge((0,2), "r", "rd", "r", "->", `poly`) // use relative coordinate names
edge((0,2), "r,rd,r", "->", `poly`) // shorthand
```

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.

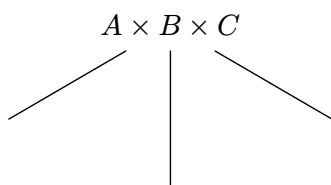


Figure 1: With defocus

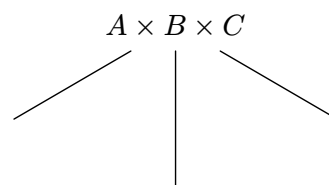
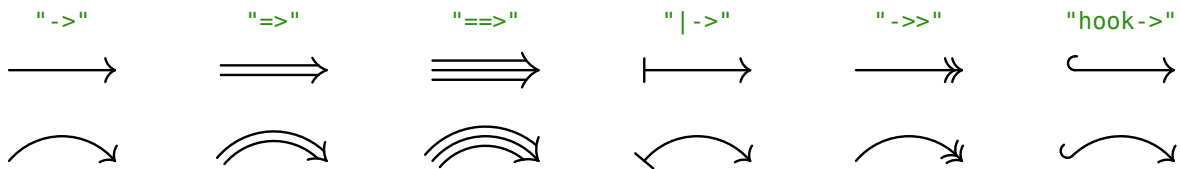


Figure 2: Without defocus

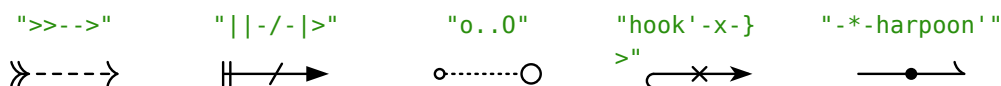
See the node-defocus argument of `diagram()` for details.

Marks and arrows

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



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



All the mark shorthands are defined in `fletcher.MARK_ALIASES` and `fletcher.MARK_DEFAULTS`:

`>`, `<`, `|`, `/`, `\`, `x`, `X`, `o`, `0`, `*`, `@`, `>>`, `<<`, `|>`, `<|`, `| |`, `>|`, `<{`, `>>>`, `<<<`, `| | |`, `bar`, `head`, `hook`, `hook'`, `hooks`, `solid`, `cross`, `circle`, `harpoon`, `harpoon'`, `doublehead`, `triplehead`

Edge styles can be specified with a shorthand like `edge(a, b, "<->")`. See the marks argument of `edge()` for details.

Adjusting marks

While shorthands exist for specifying marks and stroke styles, finer control is possible.

```
#fletcher.diagram(
  edge-stroke: 1.5pt,
  spacing: 3cm,
  edge((0,0), (-0.1,-1), bend: -10deg, marks: (
    (kind: "<>>", size: 6, delta: 70deg, sharpness: 45deg),
    (kind: "bar", size: 1, pos: 0.5),
    (kind: "head", rev: true),
    (kind: "solid", rev: true, stealth: 0.1, paint: red.mix(purple)),
  ), stroke: green.darken(50%))
)
```



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 themselves shorthands for dictionaries defining the marks' 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.
(
  size: 10.56,
  sharpness: 19deg,
  delta: 43.7deg,
  outer-len: 5.5,
  kind: "head",
)
```

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

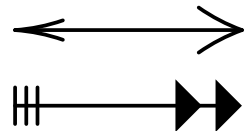
```
#fletcher.interpret-marks-arg("|=>")

// `edge(..args, marks: "|=>")` is equivalent to
// `edge(..args, ..fletcher.interpret-marks-arg("|=>"))`

(
  marks: (
    (
      size: 4.9,
      angle: 0deg,
      pos: 0,
      rev: true,
      kind: "bar",
    ),
    (
      size: 10.56,
      sharpness: 19deg,
      delta: 43.7deg,
      outer-len: 5.5,
      pos: 1,
      rev: false,
      kind: "head",
    ),
  ),
  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-solid = (kind: "solid", sharpness: 45deg)
#fletcher.diagram(
  edge-stroke: 1.4pt,
  spacing: (3cm, 1cm),
  edge((0,0), (1,0), marks: (my-head, my-head + (sharpness: 20deg))),
  edge((0,1), (1,1), marks: (my-bar, my-solid + (pos: 0.8), my-solid)),
)
```



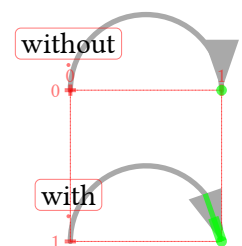
The particular marks and parameters are hard-wired and will likely change as this package is updated (so they are not documented). However, you are encouraged to use the functions `interpret-marks-arg()` and `interpret-mark()` to discover the parameters for finer control.

Hanging tail correction

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

```
#fletcher.diagram(
  edge-stroke: 2pt,
  spacing: 2cm,
  debug: 4,

  edge((0,0), (1,0), stroke: gray, bend: 90deg, label-pos: 0.1, label: [without],
    marks: (none, (kind: "solid", outer-len: 0))),
  edge((0,1), (1,1), stroke: gray, bend: 90deg, label-pos: 0.1, label: [with],
    marks: (none, (kind: "solid"))), // 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

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

Bézier edges

Currently, only straight, arc and right-angled connectors are supported. 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:

```
#fletcher.diagram(  
  node((0,1), $A$),  
  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, options)  
  
      // 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 = 0deg  
      let in-angle = -90deg  
  
      // 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, 15mm))  
      //     let c2 = (to: p2, rel: (in-angle, 30mm))  
      //     cetz.draw.bezier(  
      //       p1, p2, c1, c2,  
      //       mark: (end: ">") // cetz-style mark  
      //     )  
      //   }  
      // })  
    })  
  }  
)
```

Bézier

A

Node groups

Here is another example of how you could automatically draw “node groups” around selected nodes. First, we find all nodes of a certain fill, get their actual coordinates, and then draw a rectangle around their bounding box.

```

#let in-group = orange.lighten(60%)
#let out-group = blue.lighten(60%)

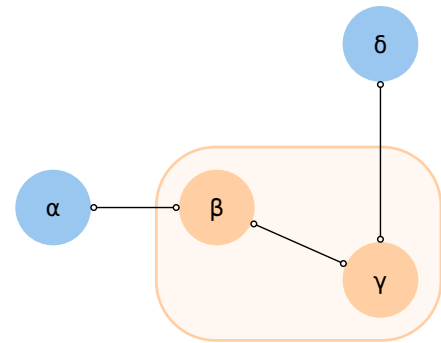
// draw a blob around nodes
#let enclose-nodes(nodes, clearance: 8mm) = {
  let points = nodes.map(node => node.real-pos)
  let (center, size) = fletcher.bounding-rect(points)

  cetz.draw.content(
    center,
    rect(
      width: size.at(0) + 2*clearance,
      height: size.at(1) + 2*clearance,
      radius: clearance,
      stroke: in-group,
      fill: in-group.lighten(85%),
    )
  )
}

#fletcher.diagram(
  node((-1,0), `α`, fill: out-group, radius: 5mm),
  edge("o-o"),
  node((0, 0), `β`, fill: in-group, radius: 5mm),
  edge("o-o"),
  node((1,.5), `γ`, fill: in-group, radius: 5mm),
  edge("o-o"),
  node((1,-1), `δ`, fill: out-group, radius: 5mm),

  render: (grid, nodes, edges, options) => {
    // find nodes by color
    let group = nodes.filter(node => node.fill == in-group)
    cetz.canvas({
      enclose-nodes(group) // draw a node group in the background
      fletcher.draw-diagram(grid, nodes, edges, options)
    })
  }
)

```



Function reference

diagram()

Draw an arrow diagram.

Parameters

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

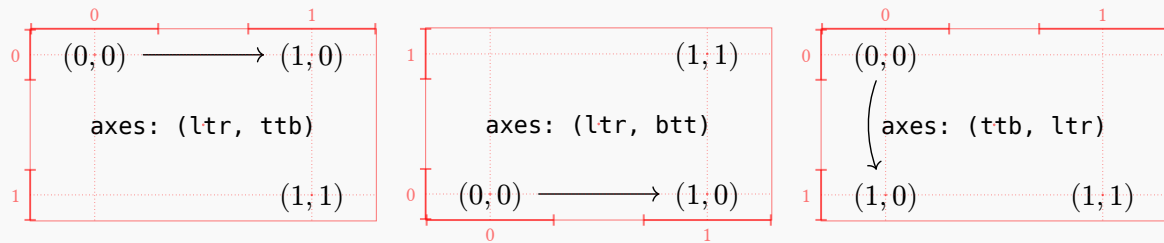
Level of detail for drawing debug information. Level 1 shows a coordinate grid; higher levels show bounding boxes and anchors, etc.

Default: false

axes pair of directions

The directions of the diagram's axes.

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



Default: (ltr, ttb)

spacing length or pair of lengths

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

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

Default: 3em

cell-size length or pair of lengths

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

Default: 0pt

edge-stroke stroke

Default value of the stroke option for edge(). By The default value for this option is chosen relative to the font size to match the thickness of mathematical arrows such as $A \rightarrow B$.

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

Default: 0.048em

node-stroke stroke or none

Default value of the stroke option for node().

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

Default: none

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

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

Default: `2.5pt`

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

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

Default: `none`

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

Default value of the inset option for `edge()`.

Default: `6pt`

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

Default value of the outset option for `edge()`.

Default: `0pt`

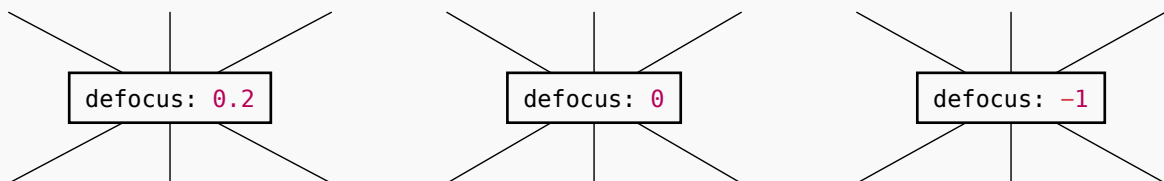
node-fill `paint`

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 `length`

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

Default: `0.2em`

mark-scale length

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

Default: 100%

crossing-fill paint

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

Default: white

crossing-thickness number

Default thickness of the occlusion made by crossing connectors. See 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(  
    ..args: any,  
    vertices: array,  
    label: content,  
    label-side: left right center,  
    label-pos: number,  
    label-sep: number,  
    label-anchor: anchor,  
    label-fill: bool paint,  
    stroke: stroke,  
    dash: dash type,  
    kind: string,  
    bend: angle,  
    corner: none left right,  
    corner-radius: length none,  
    extrude: array,  
    shift: length,  
    anchor-from,  
    anchor-to,  
    marks: pair of strings,  
    mark-scale: percent,  
    crossing: bool,  
    crossing-thickness: number,  
    crossing-fill: paint  
)
```

..args any

Positional arguments may specify the edge's:

- start and end nodes
- any additional vertices
- label
- marks

The start and end nodes must come first, and are optional:

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

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

Some named arguments, including `marks`, `label`, and `vertices` can be 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: "->")
```

Additionally, some common options are given flags that may be given as string positional arguments. These are `"dashed"`, `"dotted"`, `"double"`, `"triple"`, and `"crossing"`.

vertices `array`

Any coordinates for the edge in addition to the start and end coordinates.

These can also be positional arguments, e.g., `edge(A, D, vertices: (B, C))` is the same as `edge(A, B, C, D)`. If the number of vertices is non-zero, the edge kind defaults to "poly".

Default: `()`

label `content`

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

Default: `none`

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

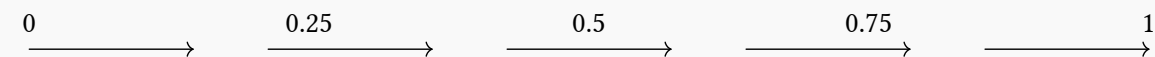
Which side of the edge to place the label on, viewed as you walk along it from base to tip. If `center`, then the label is placed directly on the edge. When `auto`, a value of `left` or `right` is automatically chosen so that the label is:

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

Default: `auto`

label-pos `number`

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



Default: `0.5`

label-sep `number`

Separation between the connector and the label anchor.

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



With `label-anchor: "center"`:



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

Default: `auto`

label-anchor anchor

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

Default: **auto**

label-fill bool or paint

The background fill for the label. If **auto**, then defaults to **true** if the label is over the edge (label-side: center). If **true**, defaults to the value of crossing-fill.

Default: **auto**

stroke stroke

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

Default: **auto**

dash dash type

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

Default: **none**

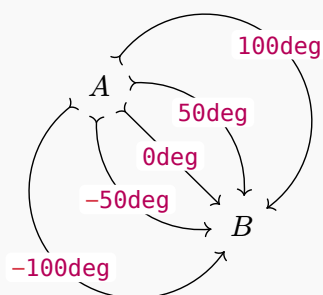
kind string

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

Default: **auto**

bend angle

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



Default: **0deg**

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

Whether to create a right-angled corner, turning left or right.

Default: `none`

corner-radius `length` or `none`

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



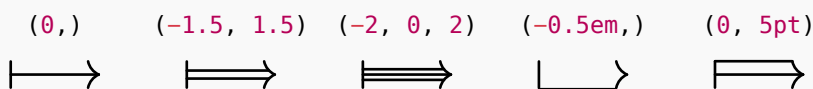
This length specifies the corner radius for right-angled bends. The actual radius is smaller for acute angles and larger for obtuse angles to balance things visually. (Trust me, it looks naff otherwise!)

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

Default: `auto`

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.

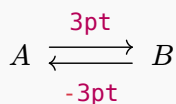


Notice how the ends of the line need to shift a little depending on the mark. For basic arrow heads, this offset is computed with `round-arrow-cap-offset()`.

Default: `(0,)`

shift `length`

Amount to shift the edge sideways by, perpendicular to its direction.



Default: `0pt`

anchor-from

Default: `auto`

anchor-to

Default: **auto**

marks pair of strings

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

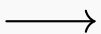
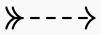
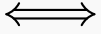
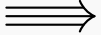
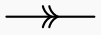
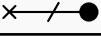
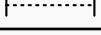
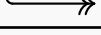
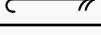
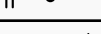
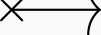
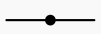
- A shorthand string such as `"->"` or `"hook' -/->"`. Specifically, shorthand strings are of the form $M_1 L M_2$ or $M_1 L M_2 L M_3$, where

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

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.

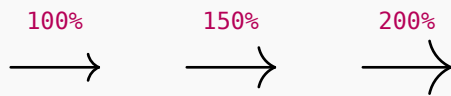
Shorthands are expanded into other arguments. For example, `edge(p1, p2, "=>")` is short for `edge(p1, p2, marks: (none, "head"), "double")`, or more precisely, `edge(p1, p2, ..fletcher.interpret-marks-arg("=>"))`.

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>

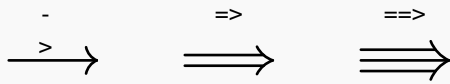
Default: `(none, none)`

mark-scale percent

Scale factor for marks or arrowheads.



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



Default: 100%

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**

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
```

Parameters

```
interpret-edge-args(  
  args,  
  options  
)
```

args

options

node()

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

Parameters

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

..args

pos point

Dimensionless “elastic coordinates” (x, y) of the node. The coordinates are usually integers, but can be fractional.

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

Default: `auto`

label content

Content to display inside the node.

Default: `auto`

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`

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`

width

Default: `auto`

height

Default: `auto`

radius

Default: `auto`

corner-radius

Default: `auto`

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

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

Some other shape functions are provided in the `fletcher.shapes` submodule, including `diamond`, `pill`, `parallelogram`, `hexagon`, and `house`.

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

- The `node` argument is a dictionary containing the node's attributes, including its center position (`node.real-pos`), the label's dimensions (`node.size`), and other options (such as `node.corner-radius`, which may not have an effect for some shapes).
- 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 node `outset`, and to create multi-stroke effects using `extrude`.

Default: `auto`

defocus `number`

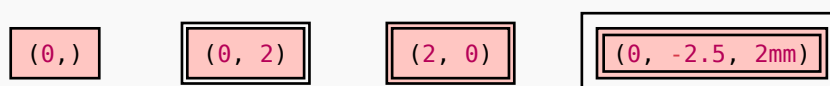
Strength of the “defocus” adjustment for connectors incident with this node. If `auto`, defaults to the `node-defocus` option of `diagram()`.

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,)`

Marks

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. `"solid"` 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.

Behind the scenes

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.



lerp-at()

Linearly interpolate an array with linear extrapolation at bounds

If the index `t` is fractional, adjacent values are linearly interpolated, and if the index is out of array bounds, the value is linearly extrapolated from the nearest two points. (This is kind of funky, but it's the padding style I wanted for coordinates going off-grid.)

Parameters

```
lerp-at(  
  a,  
  t  
)
```

a

t

compute-grid()

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

Parameters

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

nodes

edges

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 (center: (x, y), size: (width, height)). The coordinates x and y may be floats.

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.

Parameters

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

objects cetz 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.

callback