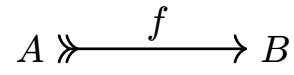


fletcher

(noun) a maker of arrows

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

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

Last updated 2026-09-17.

`fletcher-package.github.io`

Version 0.6.0

Manual

Function Reference

Manual

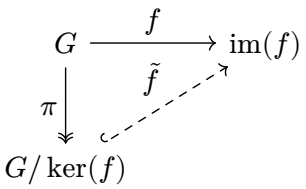
Quick Overview

Import fletcher with:

```
#import "@preview/fletcher:0.6.0" as fletcher: diagram, node, edge
```

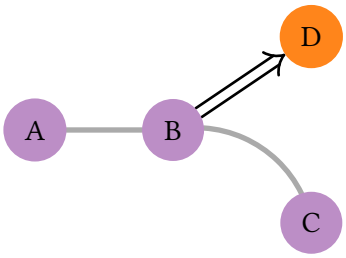
Diagrams contain **node()**s and **edge()**s. Nodes contain content and can have various **shapes** and **styles**, while edges snap to nodes and can be given **marks** and **labels**. Nodes and edges can be placed in a **diagram()** or **directly into a CeTZ canvas**. Objects in a **diagram()** are arranged on a **flexible coordinate grid**.

```
#diagram({
  node((0,0), $G$)
  edge("->", $f$) // start/end at prev/next node
  node((1,0), $im(f)$)
  edge((0,0), "->>", $pi$) // start given; end is next node
  edge("<--hook'", $tilde(f)$)
  node((0,1), $G slash ker(f)$)
})
```



Styles can be controlled with the named arguments of **node()** and **edge()**. Default styles can be set by passing the same arguments prefixed with "`node-`" or "`edge-`" to **diagram()**.

```
#diagram(
  spacing: (1cm, 4mm), // column and row gutter
  node-inset: 8pt, // padding around node content
  node-fill: teal.mix(fuchsia),
  edge-stroke: 2pt + gray,
  node((0,0))[A], edge(), node((1,0))[B],
  node((2,1))[C], node((2,-1), fill: orange)[D],
  edge((1,0), (2,1), bend: 45deg),
  edge((1,0), "=>", (2,-1), stroke: 1pt + black),
)
```



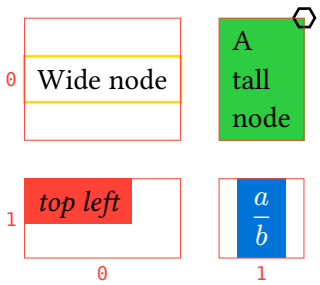
You can also place nodes and edges directly into `cetz.canvas()` and set node and edge styles with `cetz.draw.set-style()`. Below, we use fletcher's marks to draw an edge in a CeTZ-based figure.

```
#import fletcher.cetz // import this way to ensure compatibility
#cetz.canvas(length: 5mm, {
  import cetz.draw: *
  set-style(fill: teal, edge: (stroke: 1pt + teal))
  circle((5,0), radius: (1.2,0.8), name: "egg")
  line((0,1), (1,0), (0,-1), (-1,0), close: true, name: "jewel")
  edge(<jewel>, "stealth--stealth", <egg>, bend: 45deg)
})
```

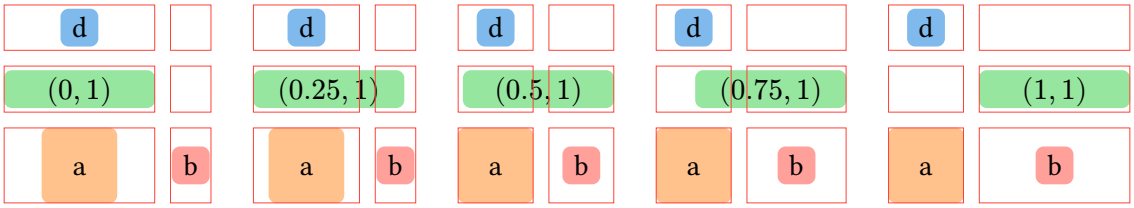
Flexible coordinate grids

Diagrams are laid out on a *flexible coorginate grid*, visible when the `debug` option of **`diagram()`** is on. The rows and columns in this coordinate system grow to accommodate the sizes of nodes, useful for tabular layouts:

```
#diagram(
  debug: "grid", // show helper annotations
  spacing: 5mm,  // gutter between cells
  node((0,0), stroke: yellow, [Wide node]),
  node((1,0), fill: green, [A\ tall\ node], name: "tall"),
  node((0,1), fill: red, emph[top left], align: top + left),
  node((1,1), fill: blue, text(white, $ a/b $)),
  cetz.draw.polygon("tall.north-east", 6, radius: 4pt)
)
```

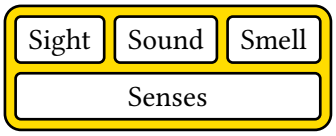


Unlike a table, coordinates can be fractional; the center of a node placed at `0.25` is 25% between the adjacent columns or rows. Notice how the column sizes respond to the green node:



When inside a flexigrid, nodes can also span multiple columns or rows:

```
#diagram(
  spacing: 4pt,
  node-stroke: 1pt,
  node-fill: white,
  node-corner-radius: 2pt,
  node((0,0), [Sight], name: <sight>),
  node((1,0), [Sound]),
  node((2,0), [Smell], name: <smell>),
  node((0,1), [Senses], colspan: 3),
  node((0,0), rowspan: 2, colspan: 3,
    fill: yellow, extrude: 4pt, layer: -1, []),
)
```

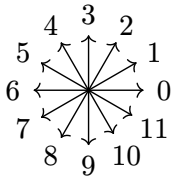


Under the hood, a **`diagram()`** is just a **`flexigrid()`** inside a CeTZ canvas. You can also use the flexigrid coordinate system inside a normal CeTZ canvas (see **CeTZ Integration** for details.)

Coordinate expressions

Dimensionless coordinates like `(2, 3)` in a **`node()`** or **`edge()`** refer to columns/rows in the grid. You can also use physical lengths like `(20mm, 5mm)` or any CeTZ coordinate expression, including anchors.

```
#diagram(
  node((1, 0), name: "0"), // elastic coordinate
  for i in range(12) {
    let θ = i/12*360deg
    edge(<0>, "->", auto)
    node((rel: (θ, 10mm), to: "0"), $ #i $, inset: 4pt)
```



```
  }
)
```

Diagrams and Layout

Fletcher encourages the use of an "elastic, tabular layout" called a *flexigrid*.

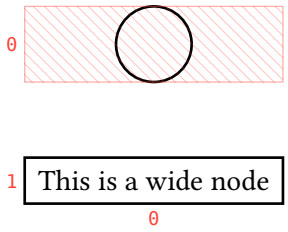
Diagrams use a flexigrid layout by default, but you can also draw in a CeTZ canvas instead (see **Drawing in a CeTZ canvas**). To render a diagram, fletcher first collects the row/column positions (or uv coordinates) of all nodes in the diagram and calculates the row and column sizes for the flexigrid layout. After the final flexigrid is determined, all nodes, edges and other objects are drawn in a context where both uv coordinates and normal xy coordinates can be used.

If nodes have fractional uv coordinates, an iterative algorithm is used to calculate the minimum row and column sizes which accommodate the nodes. This usually requires only a few steps (see the **`flexigrid.max-layout-iterations`** option for details).

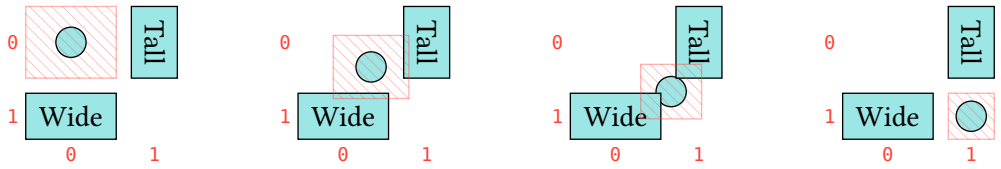
Nodes are placed within *cells*

A node inside a flexigrid lives within a *cell*, which is visible when the **`"node.cell"`** debug option is turned on for the node or diagram.

```
#diagram(
  debug: "grid.coords",
  node-stroke: 1pt,
  node((0,0), radius: 5mm, debug: "node.cell"),
  node((0,1))[This is a wide node],
)
```



Nodes placed at fractional coordinates still live in their own cell, defined by a linear interpolation. For example, below the circle's cell is shown as it moves along $(0,0) \to (1,1)$.

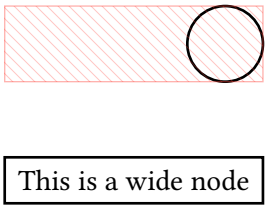


This behaviour guarantees that a small nudge in a node's position only results in a small change to the diagram's final appearance.

Node alignment within cells

A node's cell is defined by the flexigrid's rows and columns, and grows with the size of the node. However, the cell can be larger than the node's bounding box (visible with the **`"node.bounds"`** debug option). By default, nodes are placed in the center of their cell, but they can also be **aligned within cells** with **`node.align`**.

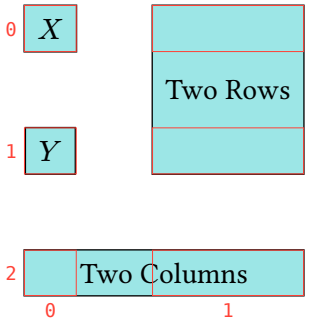
```
#diagram(
  node-stroke: 1pt,
  node((0,0), radius: 5mm, debug: "node.cell",
    align: right),
  node((0,1))[This is a wide node],
)
```



Node row and column span

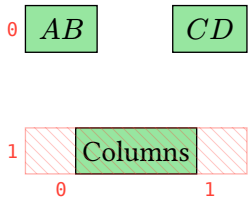
A node's cell can be made to span multiple columns or rows in a flexigrid. When this happens, the node's size is automatically set to the full size of the cell.

```
#diagram(
  debug: "grid",
  node-fill: teal.lighten(50%),
  node-stroke: 0.5pt,
  node-shape: rect,
  node((0,0), $X$),
  node((0,1), $Y$),
  node((1,0), rowspan: 2)[Two Rows],
  node((0,2), colspan: 2)[Two Columns],
)
```



Note that **`node.rowspan`** and **`node.colspan`** affect the node's enclosing *cell*, while the size of the node can be controlled independently. However, this use case is rare, since you can also place nodes in between cells by using fractional coordinates.

```
#diagram(
  debug: "grid.coords",
  node-fill: green.lighten(50%),
  node-stroke: 0.5pt,
  node((0,0), $A B$),
  node((1,0), $C D$),
  node((0,1), [Columns],
    colspan: 2, width: 16mm,
    debug: "node.cell"),
)
```



This is similar but different to **Enclose nodes**; row and column spans only work inside a flexigrid.

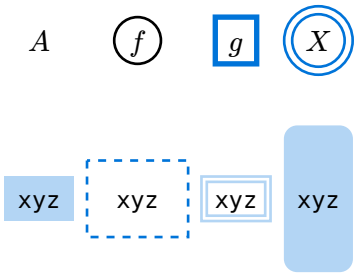
Nodes

`node(position, body, ..)`

Nodes are content centered at a coordinate. By default, nodes fit to their content (with an **inset**), but can also be given a specific size and **shape**. Nodes can be given various styles including **stroke** and **fill**.

Edges automatically snap to nodes (with an **`node.outset`**) and the positions and sizes of nodes **affects diagram layout** (unlike edges or plain CeTZ objects).

```
#diagram(
  spacing: (5pt, 2em), // small columns, large rows
  node((0,0), $A$),
  node((1,0), $f$, stroke: 1pt),
  node((2,0), $g$, stroke: 2pt + blue, shape: rect),
  node((3,0), $X$, stroke: blue, 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`, stroke: b, extrude: (0, -2))
    node((3,1), `xyz`, fill: b, height: 5em,
                       corner-radius: 5pt)
  }
)
```



Node styles

Node styles can be set with named arguments, like

`node(stroke: 2pt, ..)`

while default styles can be set by passing options to the enclosing diagram by adding a prefix, like

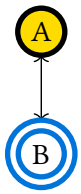
`diagram(node-stroke: 2pt, ..)`

or by using

`cetz.draw.set-style(node: (stroke: 2pt))`

which works in a **`diagram()`** and a CeTZ canvas. Like CeTZ styles, `cetz.draw.set-style()` is scoped to the current `cetz.draw.group()`.

```
#diagram(
  node-stroke: 2pt, // default stroke style for nodes
  node((0,0), fill: yellow, [A]),
  edge("<->"),
  {
    import cetz.draw: *
    set-style(node: (extrude: (2,0)))
    // stroke becomes 2pt + blue
    node((0,1), stroke: blue, [B], outset: 4pt)
  }
)
```



Available node styles:

- **`node.fill`**
- **`node.stroke`**
- **`node.extrude`**
- **`node.inset`**
- **`node.outset`**
- **`node.shape`**
- **`node.layer`**
- Any other styles specific to the **node shape**:
  - `width`, `height`, `corner-radius` for **`rect()`** nodes

- `radius` for `circle()` nodes
- **`fit` and `cell-fit`**
- and so on

Node shapes

By default, nodes are circular if their content is small and square, and rectangular if it is tall or wide. The **`shape`** option can be set to any of the following built-in shapes.

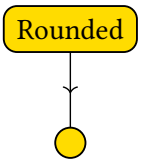
Most shapes have additional styles specific to the shape, such as:

- `width`, `height` and **`fit`** for all shapes
- `corner-radius` for `rect()`
- `radius` for `circle()`
- `angle` for `parallelogram()`, `keystone()`, `triangle()`, `house()`, `chevron()` and `hexagon()`
- `dir` for `triangle()`, `house()`, `chevron()`
- and others

Additional styles are described in each shape's documentation.

A node's shape can often be automatically inferred from the other styles given. For example, you can write `node(.., radius: 3cm)` instead of `node(.., shape: "circle", radius: 3cm)`.

```
#diagram(
  node-fill: yellow,
  node-stroke: 0.7pt,
  node((0,0), corner-radius: 5pt)[Rounded],
  edge("->-"),
  node((0,1), radius: 2mm)
)
```



Making shapes fit better

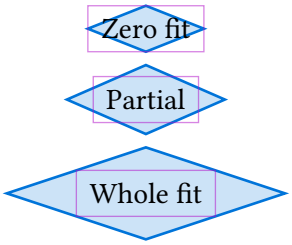
All shapes have a `fit` parameter, which adjusts how tightly the shape fits in the **`body`**.

- `fit: 0` makes the shape small enough to fit inside the body
- `fit: 1` makes the shape large enough so the node body fits inside

The default is usually in between, striking a balance. If a node looks too cramped inside a shape, you can usually adjust the `fit` instead of tweaking the **`node.inset`**.

You can see the bounding box of the node body with the **`"node.body"`** debug option.

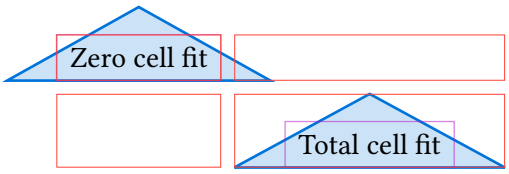
```
#diagram(
  debug: "node.body",
  spacing: 5pt,
  node-stroke: blue,
  node-fill: blue.lighten(80%),
  node-shape: "diamond",
  node((0,0), fit: 0)[Zero fit],
  node((0,1), fit: 0.5)[Partial],
  node((0,2), fit: 1)[Whole fit],
)
```



In addition to controlling how the node's body fits in the shape, the `fit-cell` parameter controls how the shape fits in the surrounding flexigrid cell.

This only matters for the layout of a surrounding **`flexigrid()`** or **`diagram()`**; the `fit-cell` style doesn't affect node itself.

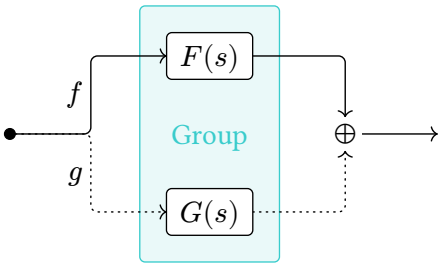
```
#diagram(
  debug: "grid.cells node.body",
  spacing: 5pt,
  node-stroke: blue,
  node-fill: blue.lighten(80%),
  node-shape: "triangle",
  node((0,0), fit-cell: 0)[Zero cell fit],
  node((1,1), fit-cell: 1)[Total cell fit],
)
```



Enclose nodes

Enclose nodes are a special type of node that are positioned around other nodes, which is useful for diagrams with nested layouts. Nodes with the **`node.enclose`** option automatically wrap around the specified nodes.

```
#diagram(
  spacing: (10mm, 5mm),
  node-stroke: 0.5pt,
  node-fill: white,
  node-corner-radius: 2pt,

  node((-2,0), radius: 2pt, fill: black),
  edge("r,u,r", "->", $f$),
  edge("r,d,r", "..>", $g$),

  node((0,-1), $F(s)$, <f>),
  edge("->", (1,0), corner: "-|"),

  node((0,+1), $G(s)$, <g>),
  edge("..>", (1,0), corner: "-|"),

  node((1,0), $ plus.o $, stroke: none, inset: 2pt),
  edge("->", "r"),

  // enclose nodes by name
  node(enclose: (<f>, <g>), text(teal)[Group],
    inset: 10pt,
    stroke: teal,
    fill: teal.transparentize(90%),
  ),
)
```



Nodes can also have **row and column spans**, which is a similar but distinct concept. Unlike nodes which span rows or columns, enclose nodes do not exist within a flexigrid layout, and do not affect the position of other nodes.

Edges

`edge(..vertices, marks, labels, ..)`

Use the **edge()** function inside a **diagram()**, `cetz.canvas()` or **flexigrid()** to draw lines or paths with various *edge effects*, including:

- **automatic snapping** to nodes or CeTZ objects
- **marks and arrows**
- **label placement**
- **corner rounding**
- **multi-stroke effects**
- CeTZ path **decorations**

You can specify an edge with a sequence of vertex coordinates, similar to `cetz.draw.line(..)`. There are a few built-in **edge kinds** which accept extra named arguments (such as `bend: 30deg` for arc edges or `corner: "|-"` for right-angled corners).

Edge effects can also be applied to any CeTZ path by **wrapping** it in the **edge()** function.

Specifying vertices

Vertices should be the first arguments, unless there are two vertices, in which case you may put the **marks argument** in between:

```
edge((0,0), (1,0), (2,1), "->")
edge(<from>, "->", <to>)
```

Alternatively, an array of vertices may be supplied to the **edge.vertices** argument.

Automatic vertices

To refer to the previous or next node in a diagram, you can use `auto`, as in:

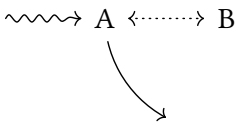
```
edge(<from>, <to>)
edge(<from>, auto)  // to next node
edge(auto, <to>)    // from previous node
edge(auto, auto)    // connects surrounding nodes
```

In unambiguous situations, `auto` vertices can be omitted altogether:

```
edge(<coord>) == edge(<coord>, auto)
edge("->", <coord>) == edge(auto, "->", <coord>)
```

For example, this diagram uses *implicit* automatic coordinates:

```
#diagram({
  edge((-1,0), "~>")               // to next node (A)
  node((0,0), [A])
  edge("<..>")                     // connects A to B
  edge("->", (.5,1), bend: -30deg) // from previous node (A)
  node((1,0), [B])
})
```

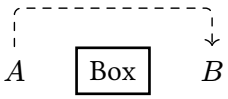


Relative coordinates

You may use strings such as `"u"` (up) or `"sw"` (south west) as shorthands for relative vertex coordinates `(rel: (du, dv))`. The first letters of **t**op/**u**p/**n**orth, **b**ottom/**d**own/**s**outh, **l**eft/**w**est, and **r**ight/**e**ast are allowed.

Commas can be used to separate multiple coordinates, so `"r,d"` is understood as `"r", "d"` which is `(rel: (1, 0)), (rel: (0, 1))`, assuming **`flexigrid.axes`** is set so (u, v) goes $(\rightarrow, \downarrow)$.

```
#diagram(
  spacing: 5mm,
  node((0,0), $A$),
  edge("u,rr,d", "-->"),
  node((1,0), stroke: 1pt)[Box],
  node((2,0), $B$),
)
```

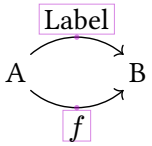


In example above, the edge **implicitly** begins from the previous node (relative coordinates cannot be used as the first coordinate).

Edge labels

Edges can have any number of labels attached to them at specific positions. Any markup or math content passed to **`edge()`** after vertices is interpreted a label.

```
#diagram(
  debug: "edge.label",
  node((0,0), [A]),
  edge("->", [Label], bend: 60deg),
  edge("->", label: $f$, bend: -60deg),
  node((1,0), [B]),
)
```



The following edge options set properties of the edge's label(s):

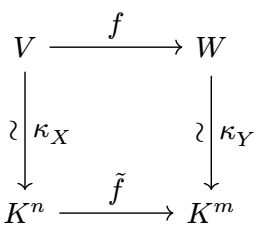
- **`label`**: the body content
- **`label-pos`**: a **path anchor** specifying the label's position
- **`label-side`**: which side of the edge to place the body
- **`label-angle`**: rotation/orientation of the body
- **`label-sep`**: separation between edge and body
- **`label-anchor`**: the CeTZ anchor to use for label body

To specify multiple labels, pass an array of dictionaries to **`edge.label`**, where each contains properties without the `label-` prefix:

```
label: (
  (body: [First label], pos: .., side: ..),
  (body: [Second label], angle: .., sep: ..),
)
```

In the example below, the vertical edges have two labels each:

```
#diagram(
  spacing: 15mm,
  node((0,0), $V$),   edge("->", $f$),        node((1,0), $W$),
  node((0,1), $K^n$), edge("->", $tilde(f)$), node((1,1),
$K^m$),

  edge((0,0), (0,1), "->", label: (
    (body: $ tilde $, angle: auto),
    (body: $kappa_X$, side: right),
  ), label-sep: 3pt), // label-sep applies to both labels

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



```
    (body: $ tilde $, angle: auto),
    (body: $kappa_Y$, side: right),
  ), label-sep: 3pt),
)
```

Kinds of edges

To make it easy to achieve common edge shapes, like arcs, loops or right-angled corners, edges can have different *kinds*, depending on the combination of named arguments present.

Edge Kind	Arguments
`arc`	`bend`
`bezier-cubic`	`from`, `to`
`bezier-from`	`from`
`bezier-to`	`to`
`bezier-through`	`through`
`loop`	`loop`, `loop-angle`
`corner`	`corner`

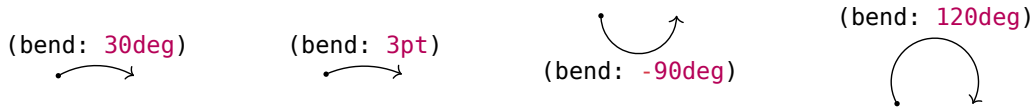
Polyline

By default, edges are displayed as straight paths between two or more vertices. As with all edge kinds, they can have **rounded corners**.

Arc

`edge(.., bend: angle | length)`

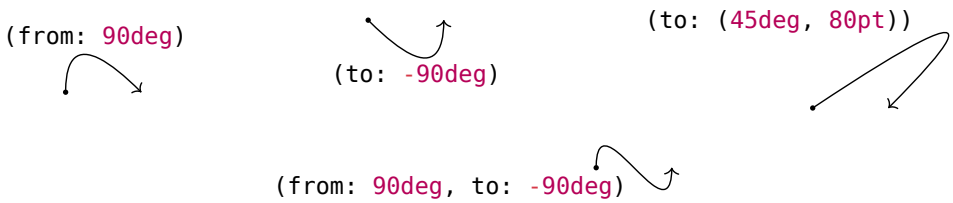
A perfect arc can be made with the `bend` option, which can be an angle (measuring the initial angle of the arc relative to a straight edge) or a length, specifying the height of the arc.



Bézier

`edge(.., from: angle | (angle, length), to: angle | (angle, length))`

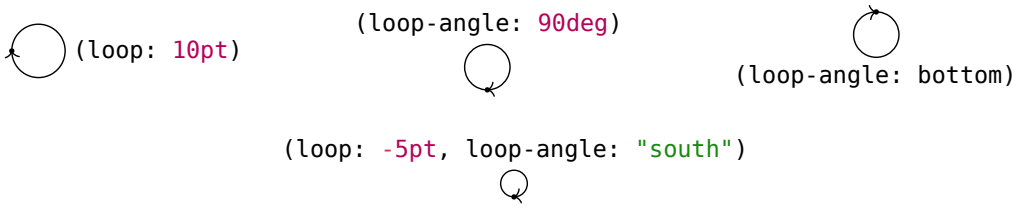
Quadratic or cubic Bézier curves can be specified by giving an angle or polar coordinate such as `(90deg, 5mm)` to one of `from`, `to` or both.



You can also specify a Bézier curve through another point with `edge(.., through: <coord>)`.

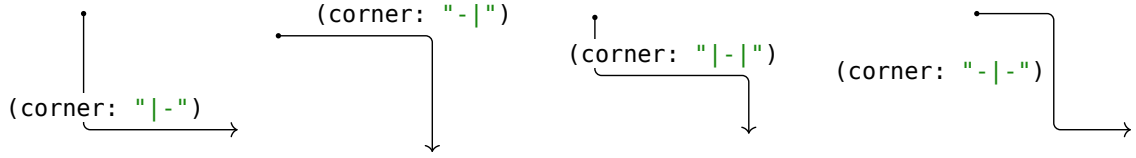
Loop

A perfectly circular loop can be specified by giving either the loop's radius as `loop` or its direction `loop-angle` (which can be an angle or a direction like `"north"` or `right`.)



Corner

Edges with one or two right-angled corners can be specified with `corner`, which is a string of `"-"` and `"|"` specifying the order of horizontal or vertical segments.

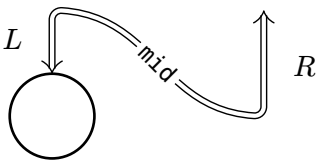


CeTZ

When **integrating with CeTZ**, you can wrap a CeTZ element in **edge()** to apply any of fletcher's edge effects to it. When used in this mode, edges may have no vertices.

For example, below we draw a composite CeTZ path using lines and a cubic Bézier segment and apply fletcher's marks, multistroke effects, label placement and snapping.

```
#cetz.canvas({
  import cetz.draw: *
  let path = merge-path({
    line((0,0), (0,1))
    bezier((0,1), (2,0), (1,1), (1,0))
    line((2,0), (2,1))
  })
  scale(1.4)
  circle((0,0), radius: .4, name: "orb")
  edge(path, "<=>", snap-to: ("orb", none), label: (
    (body: $L$, pos: 0.5),
    (body: $R$, pos: 2.5, side: right),
    (body: `mid`, pos: 1.5, side: center, angle: auto),
  ), label-sep: 10pt)
})
```



Path anchors

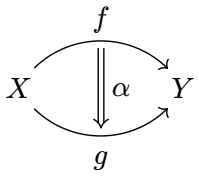
Edges support *path anchors*, like most CeTZ elements. This allows you to refer to points along an edge. In particular, the **label position** is a path anchor, and if the edge has a **`edge.name`**, you can place other elements using the coordinates `<name.anchor>`, `"name.anchor"` or `(name: "name", anchor: "anchor")`.

Path anchor	Description

`25%`	Fraction of total length along path
`"start", "mid", "end"`	Aliases for `0%`, `50%` and `100%`
`10pt, -2em`	Certain length from start or end of path
`0.5, 2.5, -1`	Segment indices referring to or between vertices of the path

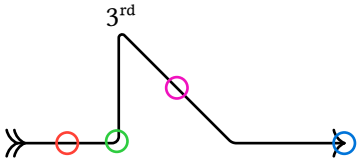
The default path anchor is the midpoint, so in the example below `<f>` refers to `(name: "f", anchor: 50%)`.

```
#diagram(
  spacing: 15mm,
  node((0,0), $X$),
  edge("->", bend: +60deg, $f$, name: <f>),
  edge("->", bend: -60deg, $g$, name: <g>),
  node((1,0), $Y$),

  edge(<f>, "=>", <g>, $alpha$, shorten: 3pt),
)
```



Fletcher defines *segment indices* in addition to CeTZ's usual path anchors, to make it easier to place labels along complex paths. A segment index is a number which interpolates between the edge's vertices. For example, `0` is the start, `1` is the first vertex, and `2.5` is halfway along the third segment of the edge. Negative indices refer to vertices in reverse order.

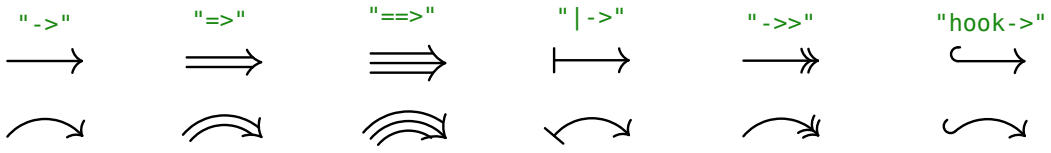
```
#diagram(
  spacing: 15mm,
  import cetz.draw: *,
  set-style(circle: (radius: 4pt)),
  edge("r,t,rd,r", ">>->", name: "foo", stroke: 1pt,
    [3#super[rd]], label-pos: 2),     // label at 3rd
vertex
  circle("foo.0.5", stroke: red),     // 50% along first
segment
  circle("foo.1", stroke: green),     // second vertex
  circle("foo.-0", stroke: blue),     // last vertex
  circle("foo.2.5", stroke: fuchsia), // 50% along third
segment
)
```



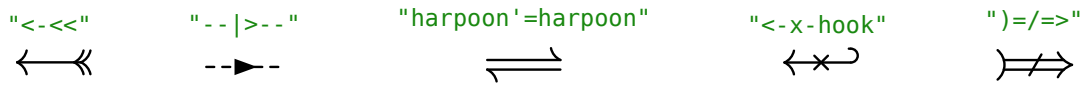
Importantly, **node positions cannot depend on edge anchors**. This is because nodes are processed before edges. To place annotations on edges, you can use labels or draw directly with CeTZ, like in the example above.

Marks and Arrows

Arrow marks may be specified like `edge(from, "->", to)` or `edge(..pts, "->")` or with the **`edge.marks`** option. Some mathematical arrow heads are supported which match the symbols →, ⇒, ⟹, ↦, ↠, and ↪ in the default font.



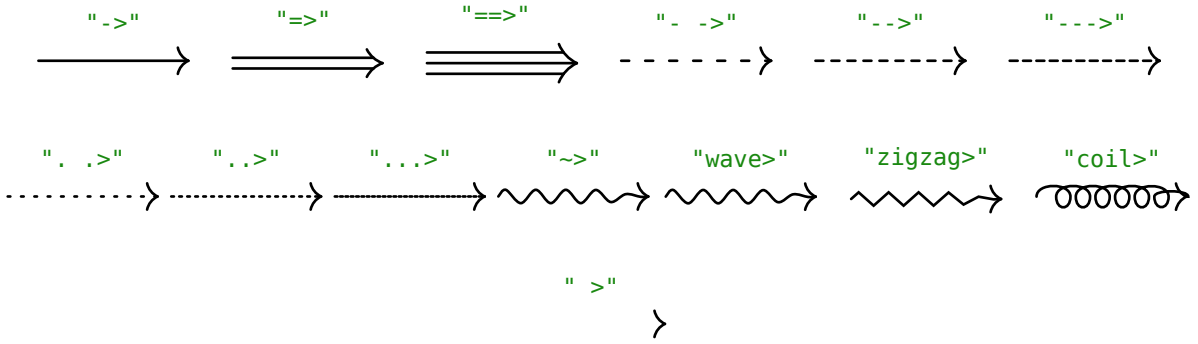
A few other built-in marks are provided, and all marks can be placed at any position along an edge.



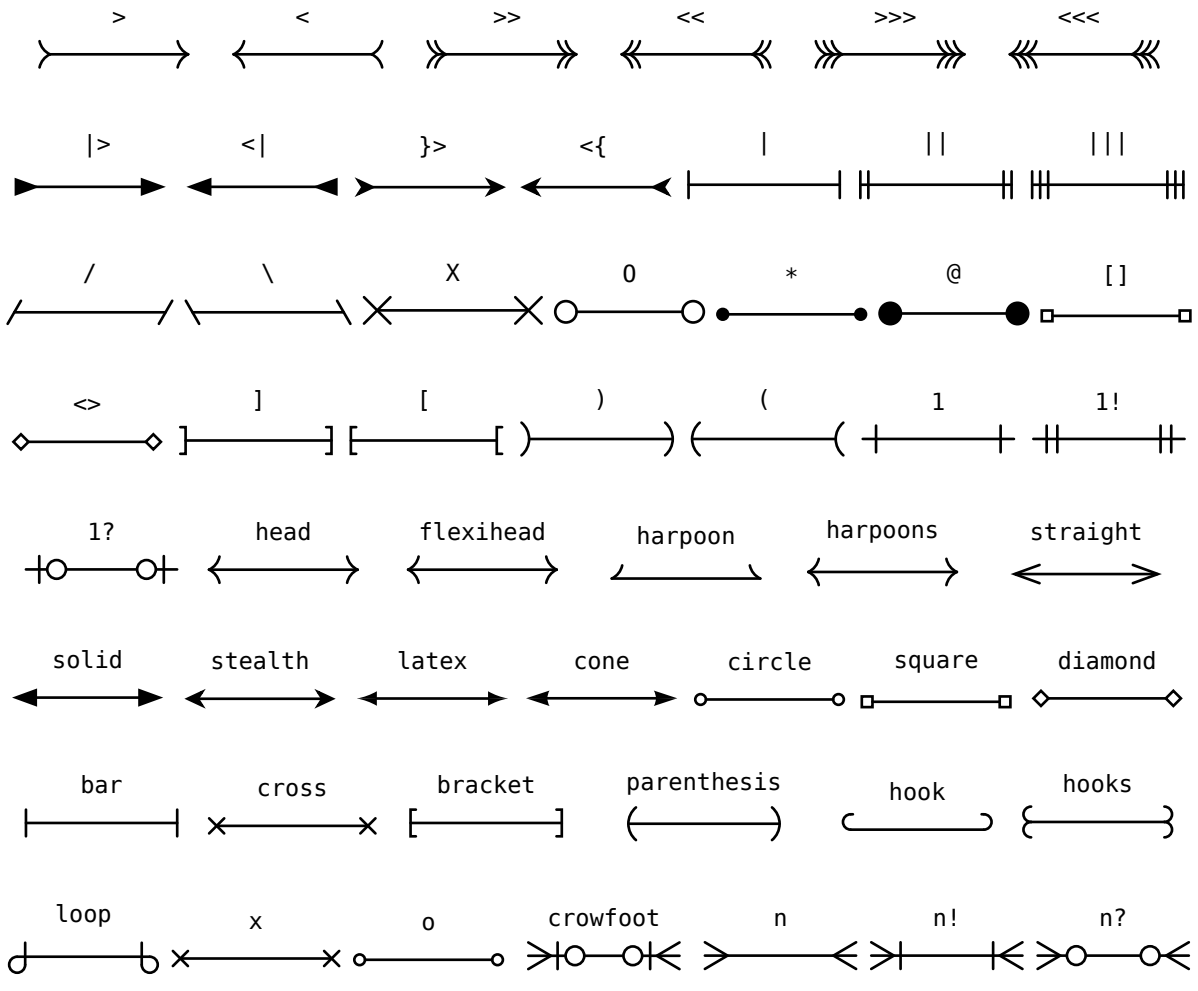
Built-in marks and line styles

A mark shorthand such as `"<->"` consists of *mark names* `"<"` and `">"` joined with *line styles*, `"-"`.

The built-in line styles are:



It is possible to rename, redefine, and define your own marks, but the built-in marks are:

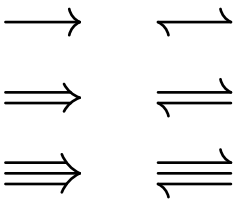


Marks can be flipped by appending `'` to the name.

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


Some marks adapt to the edge's **extrusion**, including `">"`, `"harpoon"`, `"harpoons"`, `"hook"` and `"hooks"`:

```
#diagram(
  edge-stroke: 1pt,
  edge((0,0), (1,0), "->"),
  edge((0,1), (1,1), "=>"),
  edge((0,2), (1,2), "==>"),
  edge((2,0), (3,0), "harpoon'-harpoon"),
  edge((2,1), (3,1), "harpoon'=harpoon"),
  edge((2,2), (3,2), "harpoon'==harpoon"),
)
```



Adjusting marks

While shorthands like `"--|>"` exist, finer control is possible. Under the hood, shorthands are expanded into *full form*: for example, `edge("--|>")` is the same as `edge(marks: (none, "|>"), options: (dash: "dashed"))`. This expansion is done by **`parse-mark-shorthand()`**:

```
#fletcher.parsing.parse-mark-
shorthand("--|>")
```

```
(marks: (none, "|>"), options: (dash: "dashed"))
```

For more control, you can use the full form and pass an array of marks to the **`edge.marks`** argument. This lets you pass ***mark objects*** instead of mark names, which are dictionaries of mark parameters which you can customise. For example, here is a tulip made of arrows:

```
#diagram(
  edge-stroke: 1.5pt,
  edge((0,3), (-0.1,0), bend: -8deg, marks: (
    (inherit: ">>", size: 6, delta: 70deg, sharpness: 65deg),
    (inherit: "head", rev: true, pos: 0.8, sharpness: 0deg, size: 17),
    (inherit: "bar", size: 1, pos: 0.3),
    (inherit: "solid", size: 12, rev: true, stealth: 0.1, fill:
red.mix(purple)),
  ), stroke: green.darken(50%)),
)
```

In this example, mark objects are based off previously defined marks (using the `inherit` parameter) with other parameters customised.

Listing mark parameters

You can see the parameters of built-in marks by inspecting the mark object which is stored in the `fletcher.marks.DEFAULT_MARKS` dictionary:

```
`>` = #fletcher.marks.DEFAULT_MARKS.at(">")
```

```
> = (inherit: "flexihead", rev: false)
```

```
`head` =
#fletcher.marks.DEFAULT_MARKS.at("head")
```

```
head = (
  size: 7,
  sharpness: 24.7deg,
  delta: 53.5deg,
```

```
    tip-origin: (..) => ..,
    tail-end: (..) => ..,
    tail-origin: (..) => ..,
    tip-hang: (..) => ..,
    tail-hang: 0,
    stroke: (cap: "round"),
    draw: (..) => ..,
    cap-offset: (..) => ..,
  )
```

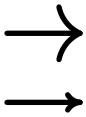
As you can see, the basic `head` mark (which `>` and `<` inherit from) has quite a few parameters:

- `size` is the overall size in units of the edge's stroke thickness
- `sharpness` is the angle formed at the tip
- `delta` is the angle of the arc spanned by the legs of the arrow
- ...and **other parameters** which are common to all marks

Tweaking mark parameters

To adjust a mark, create a mark object that inherits from the mark and overrides any parameters. For example, here is a smaller, pointerer version of the `">"` mark:

```
#let my-mark = (inherit: ">", size: 3, sharpness: 5deg)

#diagram(edge(stroke: 2pt, "->"))

#diagram(edge(stroke: 2pt, marks: (none, my-mark)))
```

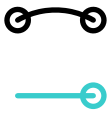


Mark objects

A *mark object* is a dictionary of parameters, which must include a `draw` entry or an `inherit` entry which points to another mark object. The `draw` entry eventually contains CeTZ objects which are translated and scaled to fit the edge; the mark should be centered at `(0, 0)` and pointing right, and the stroke's thickness is defined as the unit length.

As a minimal example, here is a basic circle mark object:

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



A mark object can contain arbitrary *parameters*, which can be adjusted to customise the mark. Any entry in a mark object can depend on parameters defined earlier by writing it as a function `mark => (..)`, where `mark` is a dictionary containing the preceding computed parameter values.

For example, our mark object from above could also be written as:

```
#let my-mark = (
  size: 2,
```

```
  draw: mark => draw.circle((0,0), radius: mark.size, fill: none)
)
```

The `size` parameter makes it easy to adjust out new mark:

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

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

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

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

Special mark properties

A mark object may contain arbitrary properties, but the following have special functions.

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

Name	*Description*	*Default*
`tip-end` `tail-end`	These control the x coordinate at which the edge's stroke terminates, relative to $(0, 0)$. See `mark-debug()`.	`0`
`cap-offset`	A function `(mark, y) => x` returning the x coordinate at which the edge's stroke terminates relative to `tip-end` or `tail-end`, as a function of the y coordinate. This is relevant for **extruded** edges. See `cap-offset()`.	

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

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

CeTZ Integration

Fletcher aims to be compatible with CeTZ, in the sense that you can use as little or as much of fletcher's features with CeTZ as you want, and vice versa. Under the hood, fletcher builds upon CeTZ by defining:

- **`flexigrid()`**, which defines a uv coordinate system for table-like layouts within a CeTZ canvas;
- **`node()`** for placing content which influences the layout of an enclosing **`flexigrid()`**;
- **`edge()`** for drawing paths with convenient styling options including outline snapping, arrow marks, multistroke effects, corner rounding, and label placement;
- **`diagram()`**, which is simply **`flexigrid()`** wrapped in a CeTZ canvas with styling shortcuts.

Drawing in a CeTZ canvas

Instead of using **`diagram()`**, you can place nodes and edges directly inside a CeTZ canvas. If you still want to use tabular layouts like in **`diagram()`**, you can wrap the objects in a **`flexigrid()`**. For example, the following are equivalent:

Using **`diagram()`**

```
#diagram(
  node-fill: teal,
  edge-stroke-thickness: 1pt,
  node((0,0), [Hello]),
  edge("->"),
  node((2,1), [There]),
)
```

Using `cetz.canvas()`

```
#cetz.canvas({
  import cetz.draw: *
  set-style(node: (fill: teal),
            edge: (stroke: 1pt))
  fletcher.flexigrid({
    node((0,0), [Hello])
    edge("->")
    node((2,1), [There])
  })
})
```

Drawing inside a CeTZ canvas is encouraged when your diagram does not have a natural table-like structure or is very complex. However, some features only work inside **`diagram()`** or **`flexigrid()`** (for example, in a `cetz.canvas()` edges cannot snap to nodes declared later).

Normal CeTZ objects may be placed in a **`flexigrid()`**, but must opt-in to use uv coordinates with `(uv: coord)`, while nodes and edges use uv coordinates by default (you can use `(xy: coord)` to opt-out). Nodes and edges can be given names and participate in CeTZ's anchoring system.

Applying edge effects to CeTZ paths

You can wrap a CeTZ path in **`edge()`** to apply edge effects to it (see **CeTZ**). Using **`edge()`** in this way is a convenient wrapper to the path modification functions offered by fletcher, which includes **`path-effect()`** for applying corner rounding (for **`edge.corner-radius`**) and offsetting (for **`edge.extrude`**) to CeTZ paths, and **`apply-edge-effects()`** for placing marks and labels.

Debugging

Fletcher has a range of *debug options* which cause extra visual feedback to be drawn in diagrams. For example, `diagram(debug: "grid")` shows a coordinate grid, and `node(debug: 4)` shows various parts of a node's anatomy.

Debug arguments accept the following options:

- `true`, `false`: enable or disable all debug annotations
- `1`, `2`, ..., `6`: enable the most common annotations up to this level of detail
- `"grid"`: enable default annotations for a category
- `"grid.cells"`: enable a specific annotation
- `("grid", "node.outset")`: multiple annotations
- `"grid node.outset"`: space-separated shorthand for multiple annotations

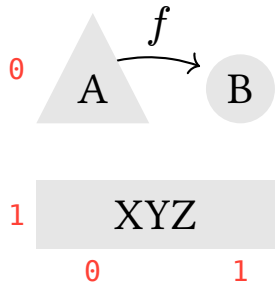
Grid debug options

Grid annotations apply to **`diagram()`** and **`flexigrid()`**.

`"grid.coords"`

Show column and row or (u, v) diagram coordinates.

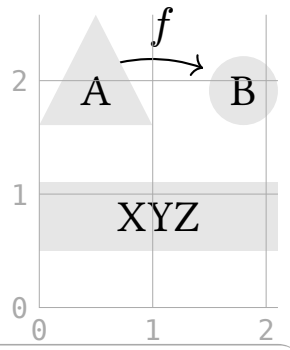
(Level ≥ 1)



"grid.xy"

Show CeTZ or (x, y) canvas coordinates.

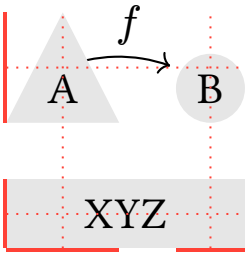
(Level $\geq$ 5)



"grid.lines"

Draw coordinate lines and indicate the sizes of rows and columns with tabs around the border.

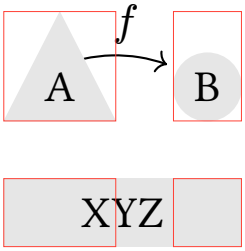
(Level $\geq$ 3)



"grid.cells"

Show all cells in a flexigrid as red boxes.

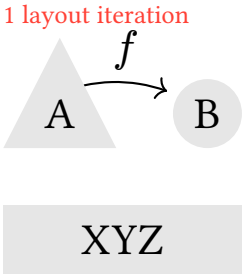
(Level $\geq$ 2)



"grid.iters"

Report the number of iterations the flexigrid layout took to converge.

(Level $\geq$ 6)



Node debug options

These can be set on individual nodes

```
node(.., debug: "node.body")
```

or set at the diagram level:

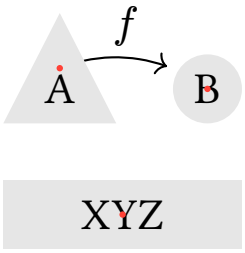
```
diagram(debug: "node.body", ..)
```

"node.origin"

Show the center coordinate of a node as a small red dot.

This point is used as the default anchor for the node.

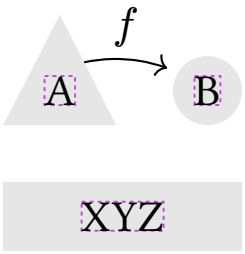
(Level $\geq$ 5)



"node.inset"

Show bounding box around the node's body content *before* **`node.inset`** is applied.

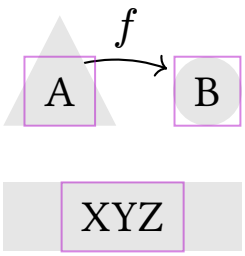
(Level $\geq$ 3)



"node.body"

Show bounding box around a node's body, including any **`node.inset`**.

(Level $\geq$ 3)

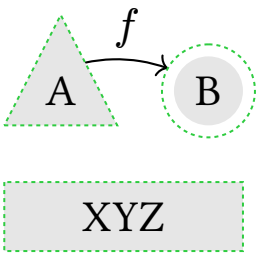


"node.outset"

Show the snapping target for edges connecting to this node.

The snapping target is the node's outline extruded by the distance **`node.outset`**.

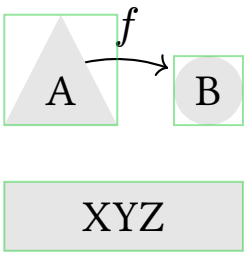
(Level $\geq$ 4)



`"node.bounds"`

Show bounding boxes of node shapes.

(Level ≥ 5)

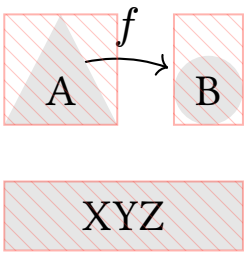


`"node.cell"`

Show the flexigrid cell inhabited by the node.

Unlike **`"grid.cells"`**, this makes the node's **column** and **row spans** visible. This node's cell is the region in which **`node.align`** has effect.

(Level ≥ 6)



Edge debug options

These can be set on individual edges

```
edge(.., debug: "edge.label")
```

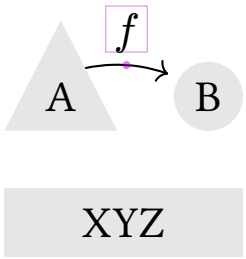
or set at the diagram level:

```
diagram(debug: "edge.label", ..)
```

`"edge.label"`

Show anchor points and bounding boxes of edge labels

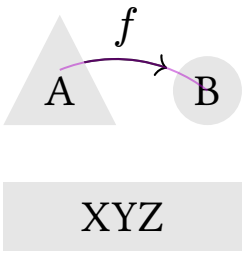
(Level ≥ 5)



`"edge.snap"`

Show the path of an edge *before* node snapping is applied

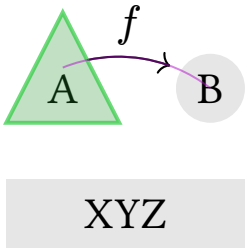
(Level $\geq$ 5)



`"edge.snap.from"`

Identify the node which the *start* of an edge snapped to by shading it green

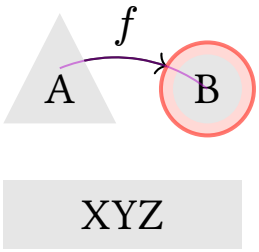
(Level $\geq$ 6)



`"edge.snap.to"`

Identify the node which the *end* of an edge snapped to by shading it red

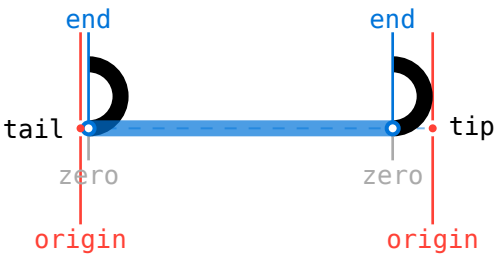
(Level $\geq$ 6)



Debugging arrow marks

To make properly implementing custom marks easier, the **`test()`** is provided which shows more detailed debug annotations on marks.

```
#show: scale.with(150%, reflow: true)
#fletcher.marks.test((
  size: 2,
  tip-origin: mark => mark.size + 0.5,
  tail-origin: -0.5,
  draw: mark => cetz.draw.arc((0,0),
    start: -90deg, stop: +90deg,
    radius: mark.size,
    fill: none,
  )
))
```



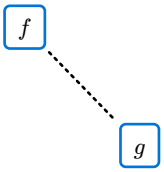
Function Reference

fletcher.diagram()

Draw nodes, edges and CeTZ objects in a `flexigrid()` layout.

Default styles for nodes and edges may be specified with named arguments such as `node-fill` or `edge-stroke-thickness`.

```
#diagram(
  node-shape: rect,
  node-corner-radius: 2pt,
  node-outset: 3pt,
  node-stroke: blue,
  edge-stroke-thickness: 1pt,
  node((0,0), $f$),
  edge(".."),
  node((1,1), $g$),
)
```




```
fletcher.diagram(..args, axes)
```

fletcher.node()

Place a *node* in a diagram or CeTZ canvas.

Nodes are content which `edge()`s can snap to. Nodes can have various shapes (rect, circle), styles (fill, stroke).

```
fletcher.node(
  ..args,
  body: content,
  shape: auto none string,
  fill,
  stroke,
  inset: length array dictionary,
  outset: length,
  extrude: array,
  layer: number,
  name: label str,
  align: alignment,
  weight: number,
  enclose: array,
  snap: bool,
  colspan: number none,
  rowspan: number,
  in-math: bool,
  debug,
)
```

`body` `content` default `none`

Content to draw in the node.

This content is measured to automatically determine the size of the node. The `"node.body"` debug option shows the body's bounding box after **`inset`** is applied.

`shape` `auto` or `none` or `string` default `auto`

The shape of the node's body enclosing its label.

Built-in shapes are `"none"`, `"rect"`, `"circle"`, `"ellipse"`, `"pill"`, `"parallelogram"`, `"keystone"`, `"diamond"`, `"triangle"`, `"house"`, `"chevron"`, `"hexagon"`, `"octagon"` and `"cylinder"`.

Some node shapes accept other styling options which can be passed as arguments to **`node()`**.

See also the **Node shapes** section of the manual.

`fill` default `auto`

Fill style of the node.

The fill is drawn within the outline defined by the first **`extrude`** value. For example:

```
#diagram(
  node-fill: yellow,
  node-stroke: 1pt,
  node((0,0), [A], extrude: (0, 3)),
  node((1,0), [B], extrude: (3, 0)),
)
```

*This option is a **node style**.*

`stroke` default `auto`

Stroke style for the node outline.

*This option is a **node style**.*

`inset` `length` or `array` or `dictionary` default `auto`

Padding applied to the content in a node's body.

The `"node.inset"` debug option draws a box around the body content before inset is applied.

The inset can be a length like `5pt`, or a CeTZ-style array or dictionary: for example, `(0, 5pt)` for only horizontal padding; `(left: 5pt, rest: 10pt)` for per-edge padding.

*This option is a **node style**.*

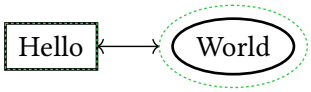
`outset` `length` default `auto`

Separation between the node's visible outline and the snapping target for edges.

This does not affect the node's appearance or layout, only how closely edges connect to it.

When `"node.outset"` debug option is on, the node outset drawn as a green dotted line.

```
#diagram(
  debug: "node.outset",
  node-stroke: 1pt,
  node((0,0), [Hello]),
  edge("<->"),
  node((1,0), [World], outset: 5pt, shape: "ellipse"),
)
```



*This option is a **node style**.*

See also **`edge.outset`**, which controls how closely individual edges connect to nodes.

`extrude` `array` default `auto`

Draw strokes around the node at the given offsets to obtain a multi-stroke effect. Offsets can 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.

*This option is a **node style**.*

`layer` `number` default `auto`

Canvas layer to draw node on.

The default layer for nodes is `0`, which is above edges (on layer `-1` by default; see **`edge.layer`**). Enclose nodes are drawn on layer `-2` by default, under nodes and edges. Nodes with equal layer are drawn in the order they are inserted.

*This option is a **node style**.*

`name` `label` or `str` default `none`

Name of the node for use with coordinate anchors.

This can also be passed as a positional argument (but then the name must be a label, not a string).

`align` `alignment` default `center + horizon`

Align a node within its associated flexigrid cell.

This only has effect when used inside a `diagram()` or `flexigrid()`.

The node's associated cell is visible when the `"node.cell"` debug option is enabled.

`weight` `number` default `1`

How much the node influences the size of flexigrid rows/columns.

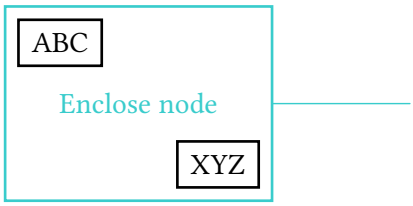
If `0`, the node does not affect the flexigrid or other node positions. If `1`, rows and columns grow to fully accommodate the node.

`enclose` `array` default `none`

Positions or names of nodes to be enclosed by this node.

When set, the node's position must be unset (or `auto`). The node is automatically positioned and enlarged so that it encloses the specified nodes.

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



`snap` `bool` default `true`

Whether this node can have edges automatically snap to it.

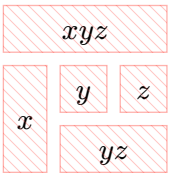
`colspan` `number` or `none` default `none`

The number of columns spanned by the node's **flexigrid cell**.

The column span can be positive (meaning the cell grows rightwards) or negative (leftwards), or even fractional. The cell must span at least one column, so the range of this parameter is $(-\infty, -1] \cup [1, \infty)$.

If the column span is not `none`, then node's width defaults to the full size of its enclosing cell.

```
#diagram(
  spacing: 5pt,
  debug: "node.cell",
  node((0,0), colspan: 3, $x y z$),
  node((0,1), rowspan: 2, $x$),
  node((1,1), $y$),
  node((2,1), $z$),
  node((2,2), colspan: -2, $y z$)
)
```



See also **`rowspan`** and **`enclose`**.

`rowspan` `number` default `none`

Row span of the node's **flexigrid cell**.

Analogous to **`colspan`**.

`in-math` `bool` default `false`

Whether to return a `metadata` object which can be placed inside equations, instead of returning an array of functions which can be inserted into a CeTZ canvas.

If you often use fletcher in math mode, consider defining a shortcut:

```
#let hom = edge.with(in-math: true)
#let obj = node.with(in-math: true)
```

Now, `hom` edges and `obj` nodes can be inserted into equations, like so:

```
#diagram($x hom(|->) & obj(pi(x), stroke: #yellow)$)
```

See also **`edge.in-math`**.

`debug` default `auto`

Enable debug annotations for only this node. See **Node debug options**.

If `auto`, the debug setting is inherited from the enclosing **`diagram()`** or **`flexigrid()`**.

`fletcher.edge()`

Draw a path with arrow marks, labels, and automatic snapping to nodes.

```
fletcher.edge(
  ..args: coord str content,
  vertices: array,
  stroke: stroke,
  dash,
  extrude: number length array,
  corner-radius: length number none,
  marks,
  mark-scale: number percent auto,
  label: content dictionary array,
  label-pos: ratio number length,
  label-side: auto none center top bottom left right start end,
  label-sep: length,
  label-fill,
  label-angle: angle auto top bottom left right,
  label-anchor: anchor,
  snap-to: none auto pair,
  snap-method: "trim" "move" pair,
  outset: length pair,
  shorten: length number array,
  name: label str,
  decorate,
  layer: number,
  crossing: bool,
  crossing-fill: color,
  crossing-thickness: number length,
  in-math: bool,
  draw: function,
  debug,
)
```

`..args` coord or str or content

An edge's positional arguments may specify:

- the edge's **`vertices`**, each given as a CeTZ coordinate;
- the CeTZ path to apply edge marks, styles, and labels to;
- the edge's **`marks`**, e.g., `"->"` or `"solid=/=solid"`;
- the body content of an edge **`label`**, e.g., `$f$`; or
- some other style flags (`dashed`, `dotted`, `double`, `triple`, `crossing`).

Vertex coordinates come first but are optional:

```
edge(from, to, ..) // explicit start and end
edge(to, ..) == edge(auto, to, ..) // start from previous node
edge(..) == edge(auto, auto, ..) // between previous and next nodes
edge(from, v1, v2, ..vs, to, ..) // multiple vertices
edge(from, "->", to) // for two vertices, marks can go in the middle
```

Vertices after the first one can be relative coordinate shorthand strings containing the characters {`l`, `r`, `u`, `d`, `t`, `b`, `n`, `e`, `s`, `w`} or commas, e.g., `edge((0,0), "u,rr,d")`.

If applying edge effects to a CeTZ path, no vertices should be given and the path should be the first argument:

```
edge(cetz.draw.bezier(..), "<->") // add marks to a cetz path
```

If given as positional arguments, an edge's **`marks`** and **`label`** are disambiguated based on their types. For example, the following are equivalent:

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

`vertices` `array` default `()`

Array of coordinates for the edge.

Vertices can also be specified as leading positional arguments (so `edge((0,1), (1,1), $f$, ..)` is the same as `edge($f$, vertices: ((0,1), (1,1)), ..)`).

`stroke` `stroke` default `auto`

Stroke style for the edge.

The default thickness matches the thickness of the → symbol in the default math font with the current text size.

The default stroke style can be set with the `edge-stroke` option of **`diagram()`** or with `cetz.draw.set-style(edge: (stroke: ..))`.

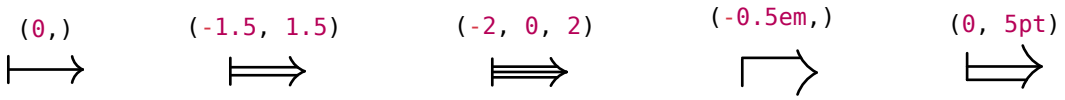
`dash` default `auto`

Set the dash property of the current stroke style.

You can also set the dash style using **`stroke`**; this is simply an alias.

`extrude` `number` or `length` or `array` default `auto`

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.

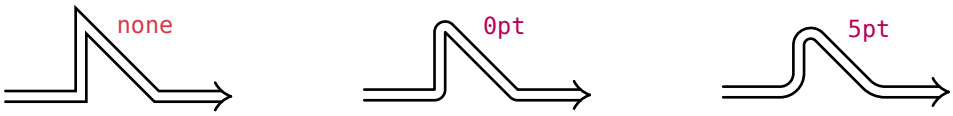


Some line styles can also be used as a shortcut:
- `edge("=")` produces `(extrude: (-2, 2))`
- `edge("==")` produces `(extrude: (-4, 0, 4))`

`corner-radius` `length` or `number` or `none` default `auto`

Radius of curvature for rounded corners.

For extruded edges, this defines the radius of curvature of the *innermost* stroke as you go around the bend. Note that `none`, which enables miter joins, is different from `0`.



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. See **`path-effect.corner-radius`** for details.

`marks` default `()`

Marks or arrows to draw along the edge.

TODO

`mark-scale` `number` or `percent` or `auto` default `auto`

Mark size multiplier.

The `size` parameter of each mark is multiplied by the mark scale before being drawn.

This is an edge style that can also be set using `diagram(mark-scale: ..)` or `cetz.draw.set-style(edge: (mark-scale: ..))`.

`label` `content` or `dictionary` or `array` default `none`

Content to place along the edge.

```
#diagram(edge("->", $f$))
```

The label body may also be given as a positional argument.

```
edge(.., [Label])
edge(.., label: [Label])
```

Label options can be specified with a dictionary, or as named arguments by adding `label-` as a prefix . For example, the following are the same:

```
edge(.., label: (body: [Label], pos: 25%))
edge(.., [Label], label-pos: 25%)
```

Possible label options are:
- `body`: the content to draw
- `angle`: orientation of the label body (see **`label-angle`**)
- `pos`: the label's position along the edge path (see **`label-pos`**)
- `sep`: padding between the label's body and the path (see **`label-sep`**)
- `side`: which side of the edge to place the body (see **`label-side`**)
- `anchor`: the CeTZ anchor to use for label body (see **`label-anchor`**)

Multiple labels can be specified with an array:

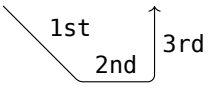
```
edge(.., label: ([First label], (body: [Second label], pos: 25%)))
```

`label-pos` `ratio` or `number` or `length` default `50%`

Position along the edge path to place labels.

This can be a `ratio`, relative to the total path length, or a `float` whose integer part refers to the segment number and whose fractional part interpolates along the segment (see **Path anchors**).

```
#diagram({
  edge((0,0), (1,1), (2,1), (2,0), "->", label: (
    (body: [1st], pos: 0.5),
    (body: [2nd], pos: 1.5),
    (body: [3rd], pos: 2.5, side: right),
  ))
})
```



This can be given as an *edge argument* like `edge(.., $f$, label-pos: 50%)` or as a **`label`** option like `edge(.., label: (body: $f$, pos: 50%))`.

`label-side` `auto` or `none` or `center` or `top` or `bottom` or `left` or `right` or `start` or `end` default `auto`

Which side of the edge to place the label on.

If `auto`, the label is placed roughly above straight edges, or on the outside of curved edges.

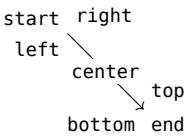
An alignment (e.g., `top`, `left`, `top + left`) means place the label beside the edge to whichever side is nearer that direction. If given as an alignment, the side may flip depending on the edge's angle.

If `true`, the label is placed above the edge assuming it goes left to right; `false` is the opposite side. In these cases, the side never flips depending on the edge's angle.

If `center` or `none`, the label is placed directly over the edge, and the label fill defaults to white.

The special alignment values `start` and `end` place the label before or after a point, travelling along the edge. This works best when used like `(pos: 0%, side: start)` or `(pos: 100%, side: end)`.

```
#diagram(edge((0,0), "->", (1,1), label: (
  (body: `start`,  side: start,  pos: 0%),
  (body: `left`,   side: left,   pos: 0%),
  (body: `right`,  side: right,  pos: 0%),
  (body: `center`, side: center, pos: 50%),
  (body: `end`,    side: end,    pos: 100%),
  (body: `top`,    side: top,    pos: 100%),
  (body: `bottom`, side: bottom, pos: 100%),
)))
```



This can be given as an *edge argument* like `edge(.., $f$, label-side: top)` or as a **`label`** option like `edge(.., label: (body: $f$, side: top))`.

`label-sep` `length` default `3pt`

Separation between label body and the edge.

This can be given as an *edge argument* like `edge(.., $f$, label-sep: 3pt)` or as a **`label`** option like `edge(.., label: (body: $f$, sep: 3pt))`.

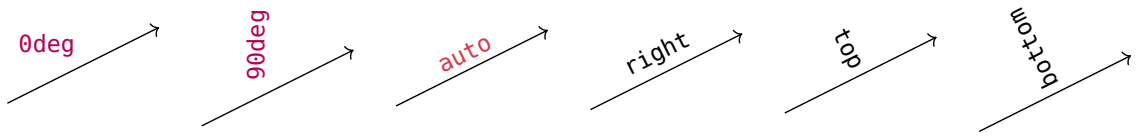
`label-angle` `angle` or `auto` or `top` or `bottom` or `left` or `right` default `0deg`

Angle of the label's body.

A positive angle goes anticlockwise, with `0deg` being upright.

An alignment (e.g., `top`, `right`) means to rotate the label with the edge's direction, such that the label is upright along edges going in that direction.

If `auto`, the best of `left` or `right` is chosen; that is, the label is rotated to be tangent to the edge and roughly the right way up.



This can be given as an *edge argument* like `edge(.., $f$, label-angle: auto)` or as a **`label`** option like `edge(.., label: (body: $f$, angle: auto))`.

`label-anchor` `anchor` default `auto`

The CeTZ anchor to use for the label content.

If `auto`, the anchor is automatically chosen depending on **`label-side`** and the edge's angle. This must be `auto` if the `side` option is set.

`snap-to` `none` or `auto` or `pair` default `(auto, auto)`

Names or coordinates of nodes or CeTZ objects to snap the edge's ends to.

This can be `none` to disable snapping or `auto` to detect nearby nodes. A pair such as `(none, auto)` can be used to control snapping at each end independently.

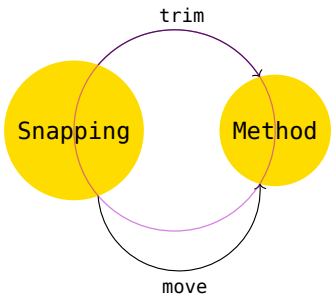
You can use the **`"edge.snap"`** debug option to see the edge's path before snapping is applied. Additionally, **`"edge.snap.from"`** and **`"edge.snap.to"`** highlight which nodes actually get snapped to.

`snap-method` `"trim"` or `"move"` or `pair` default `auto`

When an edge snaps to an object's outline, the edge can be shifted in two ways: one method is to shorten the edge to the point where it meets the object (the `"trim"` method); the other method is to move the edge's end vertex to the edge of the object (the `"move"` method).

You can pass a pair such as `("trim", "move")` to control the methods for the start and end of the edge independently.

```
#diagram(
  debug: "edge.snap",
  node-fill: yellow,
  node-shape: circle,
  node((0,0), [Snapping]),
  edge("->", `trim`, bend: +90deg, snap-method: "trim"),
  edge("->", `move`, bend: -90deg, snap-method: "move"),
  node((1,0), [Method]),
)
```

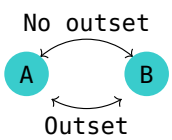


`outset` `length` or `pair` default `auto`

Gap between the end of the edge and connected nodes.

Similar to **`node.outset`**, but specific to the edge instead of the target node. Can be a single length or a pair of lengths `(from, to)` to control the outset at either end.

```
#diagram(
  node-fill: teal,
  node((0,0), [A], <a>),
  node((1,0), [B], <b>),
  edge(<a>, "<->", <b>, bend: +60deg, [No outset]),
  edge(<a>, "<->", <b>, bend: -60deg, outset: 5pt, [Outset]),
)
```



See also **`shorten`**.

`shorten` `length` or `number` or `array` default `0`

Distance to shorten the edge at either end.

If a length is given, the edge is shortened at both ends. A pair of lengths `(start, end)` controls shortening at either end of the edge independently.

See also **`outset`**.

`name` `label` or `str` default `none`

Name of the edge for use with coordinate anchors.

Giving a name to an edge allows the use of **path anchors** to connect other edges or CeTZ objects.

`decorate` default `auto`

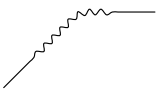
Apply CeTZ *path decorations* do the edge, such as wave or zigzag effects.

This can be a dictionary containing any of:

- `kind`, one of `"wave"`, `"zigzag"`, `"square"` or `"coil"`
- `wavelength`
- `amplitude`
- `shorten`, distance from ends to start effect from
- `smooth`, distance over which to "ramp" the effect's amplitude for a smoother transition

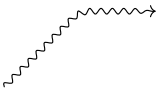
The `shorten` and `smooth` options can be lengths, distances (interpreted as multiples of `wavelength`) or a pair of these, controlling the values at the star and end of the path independently.

```
#diagram(
  edge("ru,r", decorate: (kind: "wave", shorten: 5mm, smooth: 0)),
)
```



```
#diagram(edge("rr", decorate: (
  kind: "square",
  amplitude: 3mm,
  shorten: 0,
  smooth: 2mm,
)))
```

```
#diagram(
  edge("ru,r", "~>", decorate: (shorten: (2, 0))),
)
```



`layer` `number` default `auto`

Canvas layer to draw edge on.

Edges with equal layer are drawn in the order they are inserted.

See also **`node.layer`**, which is `1` by default.

`crossing` `bool` default `false`

Draw a backdrop under the edge to give the illusion of it crossing over other lines.

If `true`, draws a backdrop of color **`crossing-fill`** with a thickness **`crossing-thickness`**, which are both styles that can be set at the diagram level.

```
#diagram(
  edge-stroke: 2pt,
  edge((0,1), (1,0)),
  edge((0,0), (1,1)),
  edge((2,1), (3,0)),
  edge((2,0), (3,1), crossing: true),
)
```

To make sure crossing lines are drawn above other lines, order them later in the diagram or use **`layer`**.

`crossing-fill` `color` default `auto`

Color of the “crossing” backdrop (drawn when **`crossing`** is enabled). This should match the background of the figure to give the illusion of breaking lines below it.

`crossing-thickness` `number` or `length` default `auto`

Width of the “crossing” backdrop (drawn when **`crossing`** is enabled) as a length or a multiple of the stroke’s thickness.

`in-math` `bool` default `false`

Whether to return a `metadata` object which can be placed inside equations, instead of returning an array of functions which can be inserted into a CeTZ canvas.

If you often use fletcher in math mode, consider defining the shortcut:

```
#let hom = edge.with(in-math: true)
```

The `hom` edge function can be inserted into equations, like so:

```
#diagram($x hom(|=>) & f(x)$)
```

See also **`node.in-math`**.

`draw` `function` default `auto`

Function accepting an array of vertices and returning the CeTZ path.

This internal argument isn't meant for normal use. It is set automatically depending on the inferred **edge kind**. For example, if `bend: 30deg` is given, `draw` defaults to `cetz.draw.arc(..)` with appropriate end points.

`debug` default `auto`

Enable debug annotations for only this edge. See **Edge debug options**.

If `auto`, the debug setting is inherited from the enclosing **`diagram()`** or **`flexigrid()`**.

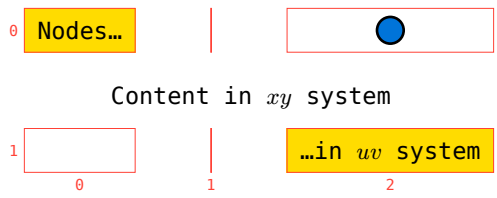
`fletcher.flexigrid()`

A "flexible" coordinate system to be placed in CeTZ canvas which adapts to nodes contained therein.

Objects drawn inside a `flexigrid` have access to a uv coordinate system, which is a nonlinear grid of rows and columns which stretch to fit nodes, like a table. Only content placed in **`node()`** can affect the uv grid; **`edge()`**s and plain CeTZ objects in a flexigrid never affect the layout.

By default, nodes and edges use uv coordinates while CeTZ objects use the default xy coordinates. Use the coordinate expressions `(uv: ..)` and `(xy: ..)` to specify the system. Both systems can be mixed in coordinate expressions like `((uv: (1,2)), 50%, (xy: (0,0)))`.

```
#cetz.canvas({
  import cetz.draw: *
  fletcher.flexigrid(debug: "grid", {
    set-style(node: (fill: yellow))
    node((0,0))[Nodes...]
    node((2,1))[...in $u v$ system]
    circle((uv: (2,0)), radius: 5pt, fill: blue)
    content((3,2))[Content in $x y$ system]
  })
})
```



The main **`diagram()`** function is essentially equivalent to **`flexigrid()`** wrapped in `cetz.canvas()`.

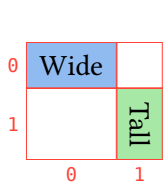
```
fletcher.flexigrid(
  ..args,
  spacing: number length pair,
```

```
  axes: array,
  max-layout-iterations: int,
  debug,
)
```

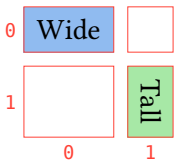
spacing `number` or `length` or `pair` default `1.0`

Gutter between cells.

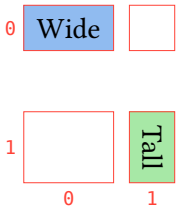
Numbers are interpreted in CeTZ units. Column and row gutter can be controlled independently as the first and last numbers in a pair, `(col-gutter, row-gutter)`.



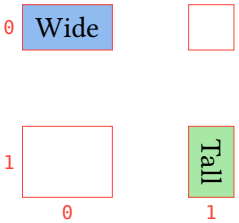
`spacing: 0`



`spacing: 5pt`



`spacing: (0.2, 0.8)`

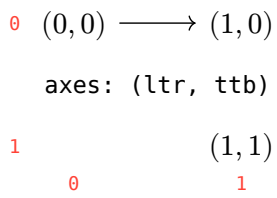


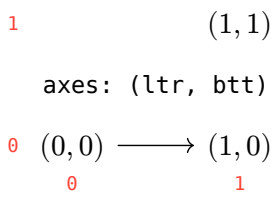
`spacing: 1`

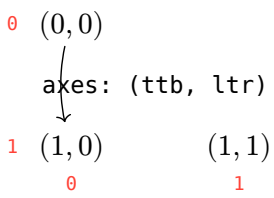
axes `array` default `(ltr, ttb)`

The physical orientation of the (u, v) axes.

This affects the elastic coordinate system used by **nodes** and **edges**. By default, the u coordinate increases $\rightarrow$ and v increases $\downarrow$. To use $(\rightarrow, \uparrow)$, which is what CeTZ uses by default, set `axes: (ltr, btt)`.







max-layout-iterations `int` default `20`

Maximum number of layout iterations used to find row and column sizes before converging.

Diagrams with nodes at fractional uv coordinates may require more iterations of the layout algorithm until the flexigrid stabilizes. You can see how many iterations were used with the `debug: "grid.iters"` option.

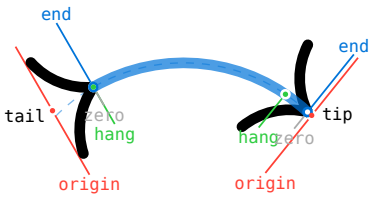
The `marks` module

`fletcher.marks.test()`

Visualise the anatomy of a fletcher mark.

This is useful for debugging mark objects.

```
#fletcher.marks.test(">", bend: 45deg)
```




```
fletcher.marks.test(
  mark,
  stroke,
  length,
  bend,
  debug,
)
```

`fletcher.marks.add-mark-defaults()`

Add the mandatory mark parameters to a mark dictionary.

See `fletcher.marks.MARK_REQUIRED_DEFAULTS`.

`fletcher.marks.add-mark-defaults(mark, defaults)`

`fletcher.marks.interpret-marks()`

Expand an array of mark specifiers into an array of mark dictionaries, ensuring `pos` and `rev` mark parameters are present. The default mark positions depends on the position of each mark and the number of marks; the first mark in the array is reversed by default.

For example, `("<", (inherit: "solid"))` is transformed into `((inherit: "head", pos: 0, rev: true, ..), (inherit: "solid", pos: 1, rev: false, ..))`

`fletcher.marks.interpret-marks(marks)`

`fletcher.marks.resolve-mark()`

Resolve all the parameters of a mark dictionary, evaluating any closures in insertion order. This also applies mark scale by premultiplying the `size` parameter.

`fletcher.marks.resolve-mark(mark)`

`fletcher.marks.draw-mark()`

Draw a mark at a given position and angle

```
fletcher.marks.draw-mark(
  mark: dictionary,
  stroke: stroke,
  origin: point,
  angle: angle,
  anchor: "zero" "origin" "end" "hang",
  as-tip,
  debug,
)
```

mark `dictionary`

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

stroke `stroke` default `1pt`

Default stroke style for the mark. The stroke's paint is used as the default fill style. If the mark itself has `stroke` entry, this takes precedence.

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

Coordinate of the origin in the mark's frame, `(0,0)`.

angle `angle` default `0deg`

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

anchor `"zero"` or `"origin"` or `"end"` or `"hang"` default `auto`

Which mark center to use as the origin. The coordinate frame origin is `zero`, and the others correspond to the mark's `tip-` and `tail-` properties, e.g., `tip-origin`, depending on whether the mark is acting as a tip (`pos: 1` and `rev: false` or `pos: 0` and `rev: true`) or a tail (`pos: 0` and `rev: false` or `pos: 1` and `rev: true`).

debug default `false`

Debug annotations to draw.

If the `mark.dots` debug option is on, the mark is annotated with colored dots indicating its centers:
- ◧●◨ (red dot): the mark's origin point, or where the tip is pointing

-  (blue dot): the end point for the paths's stroke
-  (green dot): the hang point, which determines how much to rotate marks by for curved paths

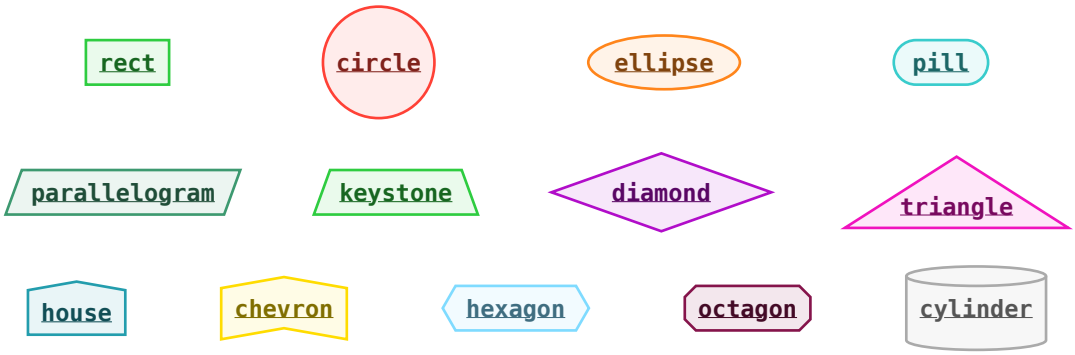
`fletcher.marks.draw-marks-on-path()`

Place marks along a path (array of subpaths), returning a dictionary containing
- `marks`: the marks to be drawn as CeTZ objects;
- `shorten-start` and `shorten-end`: the amounts that the path should be shortened from either end to accommodate terminal marks.

```
fletcher.marks.draw-marks-on-path(
  ctx,
  path,
  marks,
  stroke,
  extrude,
  debug,
) -> dictionary
```

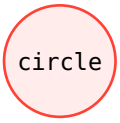
The shapes module

These are the built in **node shapes**, usable with the **`node.shape`** option.



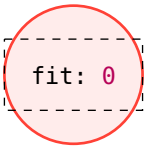
`fletcher.shapes.circle()`

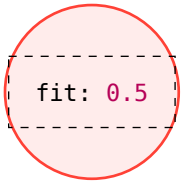
A circular node shape.

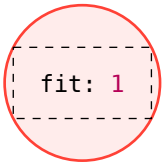



```
node(.., shape: "circle", radius: auto, fit: 0)
```

- `fit`: Adjusts how comfortably the circle fits the label's bounding box.







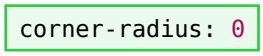

```
fletcher.shapes.circle(node)
```

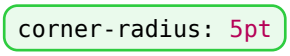
`fletcher.shapes.rect()`

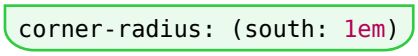
The default rectangle node shape.


```
node(.., shape: "rect", width: auto, height: auto)
```

- `corner-radius`: Accepts the same inputs as `cetz.draw.rect(radius: ..)`.







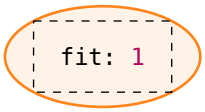

```
fletcher.shapes.rect(node)
```

`fletcher.shapes.ellipse()`

An elliptical node shape.


```
node(.., shape: "ellipse", width: auto, height: auto, fit: 0.5)
```

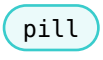
- `fit`: Adjusts how comfortably the ellipse fits the label's bounding box.




```
fletcher.shapes.ellipse(node)
```

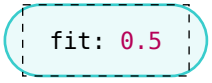
`fletcher.shapes.pill()`

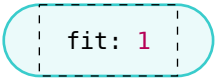
A capsule node shape.




```
node(.., shape: "pill", width: auto, height: auto, fit: 0.25)
```

- `fit`: Adjusts how comfortably the pill fits the label's bounding box.






```
fletcher.shapes.pill(node)
```

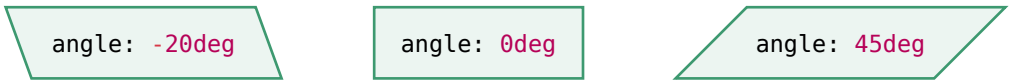
`fletcher.shapes.parallelogram()`

A slanted rectangle node shape.


```
node(.., shape: "parallelogram", width: auto, height: auto, flip: false, angle:
20deg, fit: 0.75)
```

- `flip` (boolean): Whether to slant the horizontal or vertical edges.

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

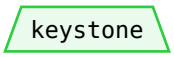


- `fit`: Adjusts how comfortably the parallelogram fits the label's bounding box.


```
fletcher.shapes.parallelogram(node)
```

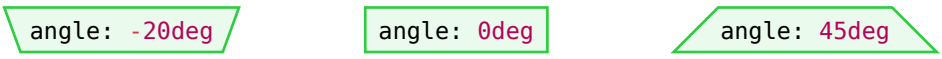
`fletcher.shapes.keystone()`

An isosceles trapezoid node shape.



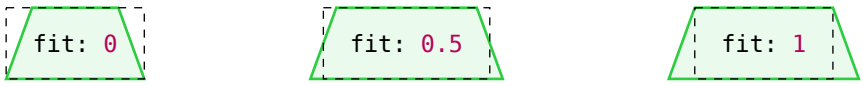

```
node(.., shape: "keystone", width: auto, height: auto, angle: 20deg, dir: top, fit:
0.75)
```

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



- `dir` (top, bottom, left, right): The side the shorter parallel edge is on.

- `fit` (number): Adjusts how comfortably the trapezium fits the label's bounding box.



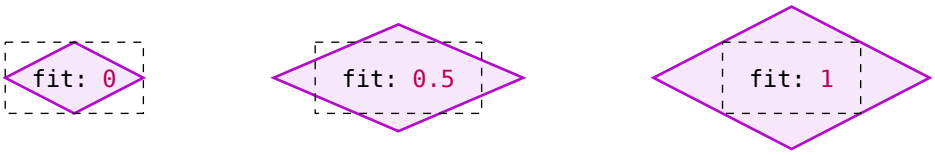

```
fletcher.shapes.keystone(node)
```

`fletcher.shapes.diamond()`

A rhombus node shape.


```
node(.., shape: "diamond", width: auto, height: auto, fit: 0.75)
```

- `fit`: Adjusts how comfortably the diamond fits the label's bounding box.




```
fletcher.shapes.diamond(node)
```

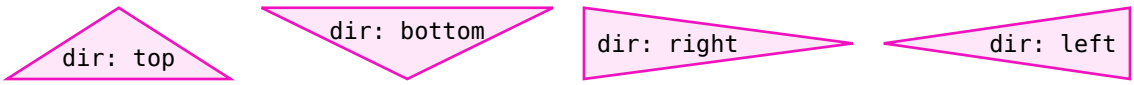
`fletcher.shapes.triangle()`

An isosceles triangle node shape.

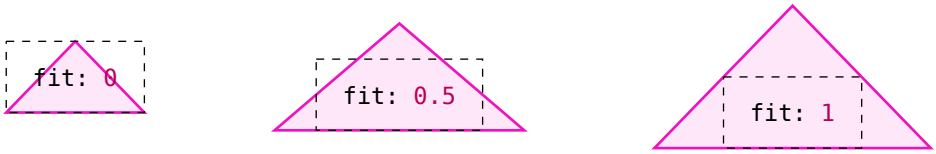

```
node(.., shape: "triangle", width: auto, height: auto, dir: top, angle: auto, aspect: auto, fit: 0.6)
```

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

- `dir` (top, bottom, left, right): The side the shorter parallel edge is on.



- `fit`: Adjusts how comfortably the triangle fits the label's bounding box.




```
fletcher.shapes.triangle(node)
```

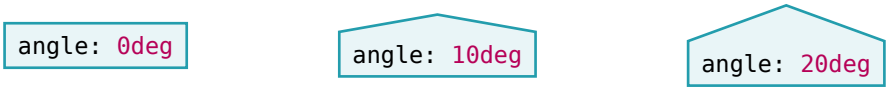
`fletcher.shapes.house()`

A pentagonal house-like node shape.


```
node(.., shape: "house", width: auto, height: auto, dir: top, angle: 10deg, fit: 1)
```

- `dir`: Direction of the roof of the house.

- `angle`: The slant of the roof. A plain rectangle is `0deg`, and `90deg` is a point stretching past Pluto.




```
fletcher.shapes.house(node)
```

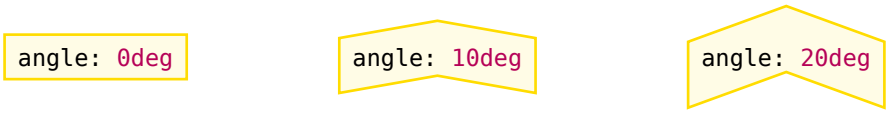
`fletcher.shapes.chevron()`

A chevron node shape.

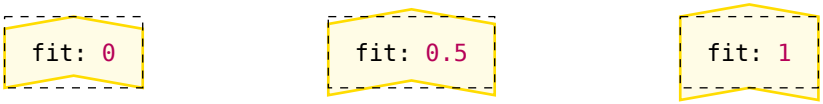

```
node(.., shape: "chevron", width: auto, height: auto, dir: top, angle: 10deg, fit: 0.8)
```

- `dir`: Direction the chevron points.

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



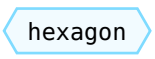
- `fit`: Adjusts how comfortably the chevron fits the label's bounding box.




```
fletcher.shapes.chevron(node)
```

`fletcher.shapes.hexagon()`

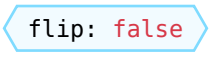
An (irregular) hexagon node shape.

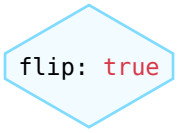



```
node(.., shape: "hexagon", width: auto, height: auto, angle: 30deg, flip: false, fit: 0.8)
```

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

- `flip` (boolean): Whether to put the points on the sides or top and bottom.





- `fit`: Adjusts how comfortably the hexagon fits the label's bounding box.

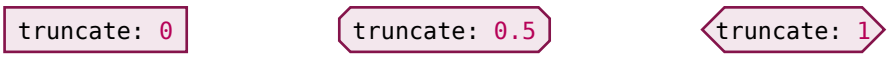

```
fletcher.shapes.hexagon(node)
```

`fletcher.shapes.octagon()`

A truncated rectangle node shape.


```
node(.., shape: "octagon", width: auto, height: auto, truncate: 0.5)
```

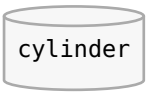
- `truncate` (number, length): Size of the truncated corners. A number is interpreted as a multiple of the smaller of the node's width or height.




```
fletcher.shapes.octagon(node)
```

`fletcher.shapes.cylinder()`

A 3D cylinder node shape.




```
node(.., shape: "cylinder", width: auto, height: auto, fit: 0.75, tilt: 8deg, rings:
(0,))
```

- `fit`: Adjusts how exactly the cylinder fits around the label's bounding box.

- `tilt` (angle): Controls the perspective tilt: `0deg` is side on.

- `rings` (length, array, none): Array of vertical positions at which to draw arcs around the body.


```
fletcher.shapes.cylinder(node)
```

The paths module

`fletcher.edges.apply-edge-effects()`

Apply edge effects to a CeTZ drawable. These effects include path extrusion and shortening, mark and label placement, and edge snapping (cutting the path at its intersections with target drawables).

```
fletcher.edges.apply-edge-effects(
  ctx,
  element,
  stroke,
  labels,
  marks,
  extrude,
  shorten,
  ..extra-path-effect-args,
  debug,
  decorate,
  crossing-stroke,
)
```

`fletcher.paths.path-effect()`

Apply path effects (extrusion and shortening) to a CeTZ object, returning a CeTZ object.

```
fletcher.paths.path-effect(
  objs: cetz objects,
  stroke,
  fill,
  shorten-start: number length array,
  shorten-end: number length array,
  extrude: number length array,
  join: "miter" "round",
  corner-radius: number length none,
  miter-limit: number,
  dynamic-radius,
)
```

`objs` `cetz objects`

CeTZ objects to apply the path effect to.

A CeTZ object is an array of functions; the result of `cetz.draw.line(..)` or `cetz.draw.merge-path(..)`, for example.

`stroke` default `auto`

Stroke style for the object, overriding the object's intrinsic stroke style.

`fill` default `auto`

Fill style for the object, overriding the object's intrinsic fill style. If `auto`, the fill is unchanged.

`shorten-start` `number` or `length` or `array` default `0`

Trim the beginning of the path by a given length.

For multi-stroke effect when **`extrude`** is an array, this may also be an array of the same length, specifying the length to shorten each offset path individually. This is useful for placing marks on multi-stroke lines correctly.

Numbers are interpreted as multiples of the stroke's thickness.

`shorten-end` `number` or `length` or `array` default `0`

Trim the end of the path by a specific length.

Works just like **`shorten-start`**.

`extrude` `number` or `length` or `array` default `0`

Lengths to offset path by. An array of different offsets results in multiple parallel strokes.

Numbers are interpreted as multiples of the stroke's thickness.

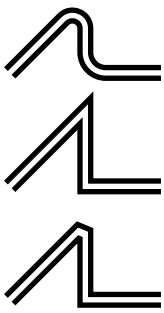
```
#import fletcher.paths: path-effect
#cetz.canvas({
  import cetz.draw: *
  set-style(stroke: 5pt)
  let obj = line((0,0), (1,1), (2,0))
  obj
  path-effect(obj, extrude: (+2, -2),
                   stroke: blue)
})
```

`join` `"miter"` or `"round"` default `"miter"`

How to form corners of an offset path.

If `"round"`, corners become rounded joints with a (minimum) corner radius specified by **`corner-radius`**. If `"miter"`, corners become miter joints or, if they are sharp enough, bevelled joints, as controlled by **`miter-limit`**.

```
#import fletcher.paths: path-effect
#cetz.canvas({
  import cetz.draw: *
  set-style(stroke: 2pt)
  let obj = line((0,0), (1,1),
                 (1,0), (2,0))
  path-effect(obj, extrude: (-1, +1),
    join: "round", corner-radius: .2)
  translate(y: -1.5)
  path-effect(obj, extrude: (-1, +1))
  translate(y: -1.5)
  path-effect(obj, extrude: (-1, +1),
    miter-limit: 2)
})
```



`corner-radius` `number` or `length` or `none` default `0`

The radius of round or bevelled corners.

For round corners, this is the radius of curvature. For bevelled corners, this is the radius of the tangent circle between the bevel face and the sides of the corner.

For extruded paths, the radii at each offset is adjusted so that the circles of curvature are concentric. At a corner, the inner paths have the specified radius of curvature, while outer paths have larger radii. The radius can be negative.

The value `none` is short for zero radius with `join: "miter"`.

Numbers are interpreted in CeTZ canvas units.

`miter-limit` `number` default `4.0`

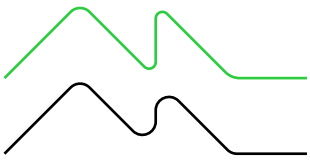
Miter limit, beyond which miter joints become bevelled.

The higher the limit, the pointier corners can be before being bevelled.

`dynamic-radius` default `true`

Whether to dynamically adjust corner radii depending on corner sharpness for nicer visual results. When enabled, the corner radius is decreased for bends of less than 90°.

```
#import fletcher.paths: path-effect
#cetz.canvas({
  let obj = cetz.draw.line((0,0), (1,1), (2,0), (2,1), (3,0), (4,0))
  let args = arguments(obj, corner-radius: 5pt, join: "round")
  path-effect(..args, stroke: green)
  cetz.draw.translate(y: -1)
  path-effect(..args, dynamic-radius: false)
})
```



`fletcher.paths.trim-path()`

Shorten a path from either end by subpath/segment index and t-parameter (not by path length, see `cetz.path-util.shorten-to` for that).

```
fletcher.paths.trim-path(
  path,
  from,
  to,
)
```

`path`

CeTZ drawable, a dictionary containing key `"segments"`.

`from` default `none`

Path index `(subpath-i, segment-i, t)` to start the path from.

`to` default `none`

Path index `(subpath-i, segment-i, t)` to terminate the path at.

`fletcher.paths.trim-to-intersection()`

Shorten a path drawable so it starts or ends at an intersection point with another drawable.

If there are multiple intersection points, the path is terminated at the one given by `index`, where intersections are ordered along the path starting from the start or end depending on the trimming mode.

```
fletcher.paths.trim-to-intersection(
  path: drawable,
  targets: array of drawables,
  trim: "start" "end",
  index: int,
)
```

path `drawable`

Drawable to truncate, of the form `(type: "path", segments: ..)`.

targets `array of drawables`

Cutting drawables which may intersect the drawable to truncate.

trim `"start"` or `"end"` default `"start"`

Whether to trim/shorten the start or end of the path.

index `int` default `0`

Index of the intersection point to use to trim the path. If trimming from the start, indexing begins at the start of the path; if trimming from the end, indexing is reversed so it starts at the intersection closest to the end.

fletcher.paths.simplify-subpath()

Simplify a subpath by deleting trivial/zero-length line segments (which end where they begin).

```
fletcher.paths.simplify-subpath(subpath)
```

fletcher.paths.cubic-second-derivative()

Get the second derivative (d^2x/dt^2) of a cubic bezier at position `t`.

- a (vector): Start point
- b (vector): End point
- c1 (vector): Control point 1
- c2 (vector): Control point 2
- t (float): Position on curve [0, 1]

```
fletcher.paths.cubic-second-derivative(
  a,
  b,
  c1,
  c2,
  t,
) -> vector
```

fletcher.paths.cubic-arc()

Approximate a circular arc with a cubic Bézier segment.

This similar to `cetz.drawable.arc()` except that it never uses more than one cubic Bézier segment, and it returns just the control and end points `(c1, c2, e)`, not a path.

Single segment approximations are useful because they are more visually robust to nudging endpoints, which is sometimes necessary when creating a rounded corner where two curves meet.

```
fletcher.paths.cubic-arc(
  x,
  y,
  z,
  start,
  stop,
  rx,
  ry,
  fill,
  stroke,
)
```

fletcher.paths.clamp-cubic-bezier()

Return the cubic Bézier obtained by clamping the parameter value t to an interval $[t_0, t_1]$.

```
fletcher.paths.clamp-cubic-bezier(
  s,
  c1,
  c2,
  e,
  t0,
  t1,
)
```

fletcher.paths.point-on-subpath-segment()

Return the position, velocity and acceleration vectors of a point at parameter value $t \in [0, 1]$ along the ith segment of a subpath.

```
fletcher.paths.point-on-subpath-segment(
  subpath: array,
  segment-index: int,
  segment-t: float,
) -> (array, array, array)
```

subpath `array`

A subpath of the form `(start: array, close: bool, segments: array)`.

segment-index `int`

The index of the subpath segment.

segment-t `float`

The time parameter of the specified segment, in the interval $[0, 1]$.

fletcher.paths.interpolate-path-point()

Given a path and an array of stops (segment indices) along the path, return the segment index of a point at a fractional stop.

The path length of stops are linearly interpolated.

```
fletcher.paths.interpolate-path-point(
  path,
  stops,
  index,
  rev,
)
```

fletcher.paths.point-on-path-by-segment()

Return the position, velocity and acceleration vectors of a point on a path at a particular segment index.

```
fletcher.paths.point-on-path-by-segment(
  path,
  index,
  rev,
)
```

index

The *segment index* is a number whose integer part refers to a segment of the path and whose fractional part specifies how far along this segment to go (in terms the linear or Bézier segment's t parameter, not its path length).

rev default `false`

Count segments from the end of the path instead of the start.

fletcher.paths.point-on-path-by-length()

Return the position, velocity and acceleration vectors of a point a specified length from the start or end of a path.

```
fletcher.paths.point-on-path-by-length(
  ctx,
  path,
  length,
  rev,
)
```

ctx

Dictionary containing a `length` key, used only to convert physical units to dimensionless CeTZ units.

length

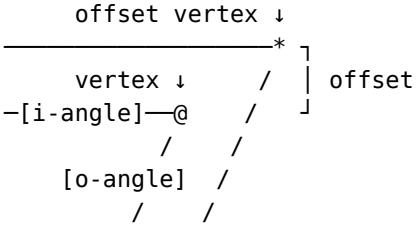
A number, length, or ratio of the path's total length.

rev default `false`

Measure from the end of the path instead of the start.

fletcher.paths.offset-vertex()

Offset a vertex to make a miter joint, given the angles of the incoming and outgoing legs.

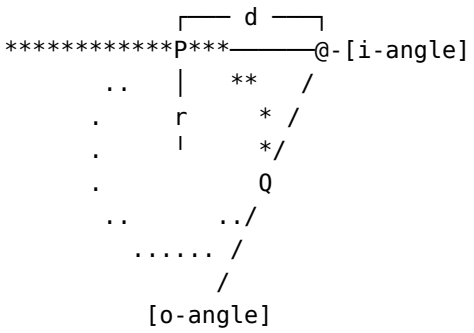



```
fletcher.paths.offset-vertex(
  vertex,
  i-angle,
  o-angle,
  offset,
)
```

`fletcher.paths.rounded-vertex()`

Incoming leg and circular segments of a rounded corner.

Returns the line segment to point `P` and a single cubic Bézier segment to `Q`.




```
@ = vertex
* = segments returned by this function
```

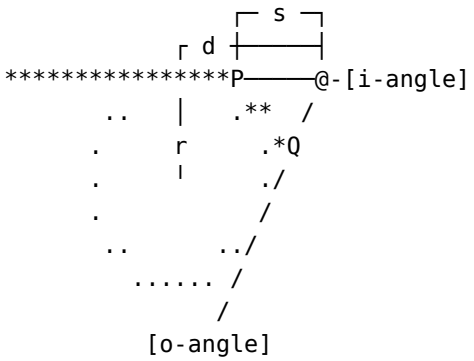
```
fletcher.paths.rounded-vertex(
  vertex,
  i-angle,
  o-angle,
  radius,
)
```

`fletcher.paths.miter-bevel-vertex()`

Segments of a miter or bevelled corner.

The bevel is specified by a radius, which defines the circle that is tangent to the bevelled face at the face's midpoint.

Depending on the miter limit, this returns a single segment for a miter join and two line segments for a bevel join, one to point `P` and the other to `Q`.




```
@ = vertex
* = segments returned by this function
```

```
fletcher.paths.miter-bevel-vertex(
  vertex,
  i-angle,
  o-angle,
  radius,
```

```
  miter-limit,
)
```

The `parsing` module

`fletcher.parsing.parse-mark-shorthand()`

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

- arg (string, array):

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

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

`fletcher.parsing.parse-mark-shorthand(arg) -> dictiony`

`fletcher.parsing.interpret-edge-positional-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
```

`fletcher.parsing.interpret-edge-positional-args(args, options)`

`fletcher.parsing.interpret-style-arguments()`

Convert named arguments such as `node-fill: yellow` into corresponding calls to `cetz.draw.set-style(node: (fill: yellow))`.

`fletcher.parsing.interpret-style-arguments(args)`