

arrow-diagrams

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

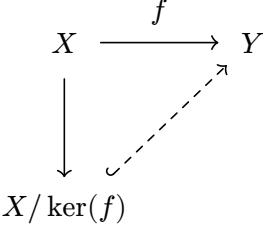
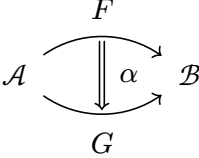
<https://github.com/jollywatt/arrow-diagrams>

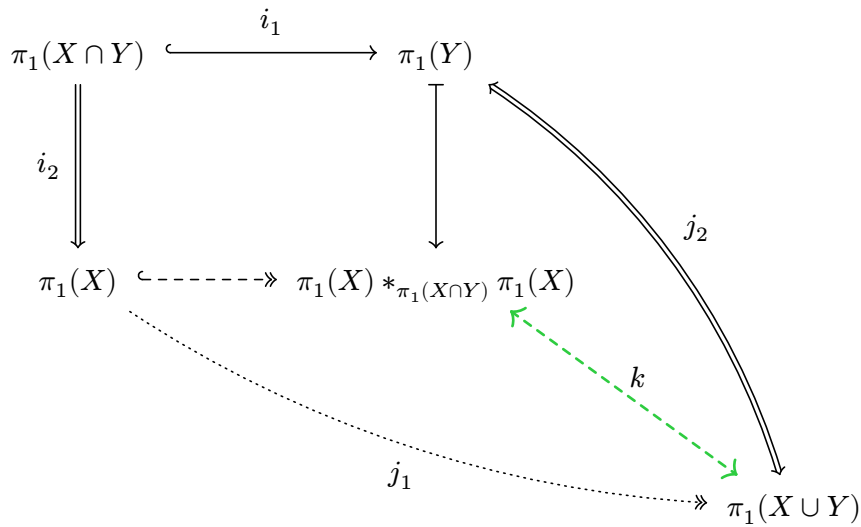
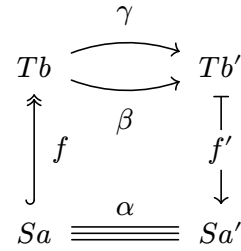
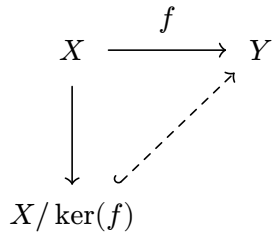
Version 0.1.0

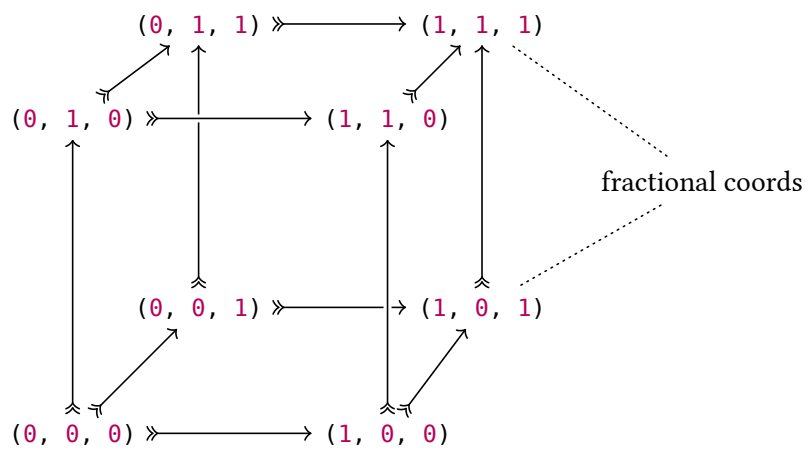
Contents

Examples	2
Details	4
Elastic coordinates	4
Physical coordinates	4
Connectors	5
The defocus adjustment	5
Function reference	6
arrow-diagram	6
conn	7
resolve-coords	11
compute-cells	11
compute-grid	12
expand-fractional-rects	12
round-arrow-cap-offset	12
get-arc-connecting-points	13

Examples

	<pre>#arrow-diagram(cell-size: 10mm, node((0,1), \$X\$), node((1,1), \$Y\$), node((0,0), \$X \text{ slash } \ker(f)\$), conn((0,1), (1,1), \$f\$, "->"), conn((0,0), (1,1), "hook-->"), conn((0,1), (0,0), "->"),)</pre>
<p>Inline $f : A \rightarrow B$ equation,</p> <p>Inline $A \xrightarrow{f} B$ diagram.</p>	<pre>Inline \$f: A \to B\$ equation, \ Inline #arrow-diagram(node-pad: 4pt, { node((0,0), \$A\$) conn((0,0), (1,0), \$f\$, "->", label-sep: 2pt) node((1,0), \$B\$) }) diagram.</pre>
	<pre>#arrow-diagram(gutter: 2cm, node((0,0), \$cal(A)\$), node((1,0), \$cal(B)\$), conn((0,0), (1,0), \$F\$, "->", bend: +35deg), conn((0,0), (1,0), \$G\$, "->", bend: -35deg), conn((.5,+.21), (.5,-.21), \$alpha\$, "=>"),)</pre>



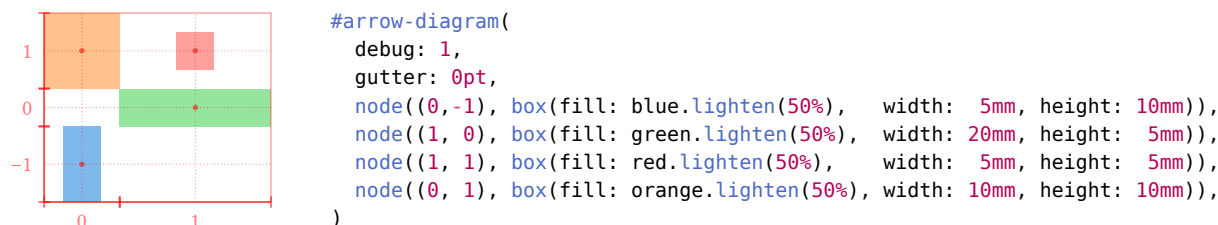


Details

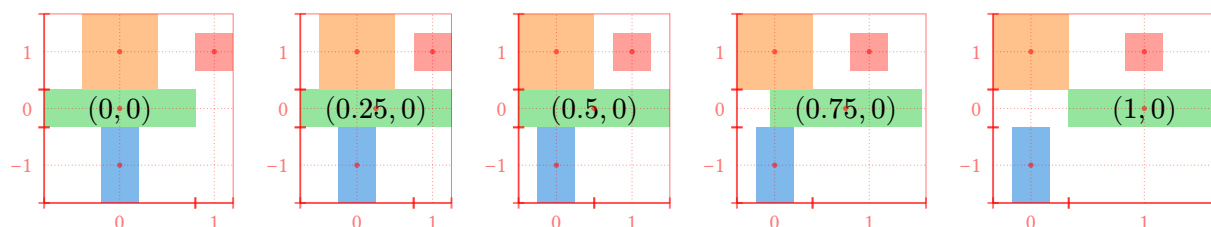
Elastic coordinates

Diagrams are laid out on a flexible coordinate grid, which stretches to fit content like a table. When a node is placed, the rows and columns grow to accommodate the node's size.

This can be seen more clearly with a coordinate grid (`debug: 1`) and no padding between cells:



While grid points are always at integer coordinates, nodes may have **fractional coordinates**. A node placed between grid points still causes the neighbouring rows and columns to grow to accommodate its size, but only partially, depending on proximity. For example, see how the column sizes change as the green box moves from (0, 0) to (1, 0):



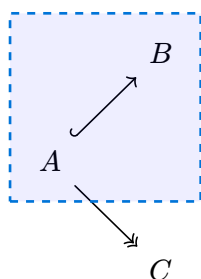
Specifically, fractional coordinates are dealt with by linearly interpolating the layout, in the sense that if a node is at (0.25, 0), then the width of column $[0.25] = 0$ is at least 75% of the node's width, and column $[0.25] = 1$ at least 25% its width.

As a result, diagrams will automatically adjust when nodes grow or shrink, while still allowing you to place nodes at precise coordinates.

Physical coordinates

Elastic coordinates are determined by the sizes and positions of the nodes in the diagram, and are resolved into physical coordinates which are then passed to CeTZ for drawing.

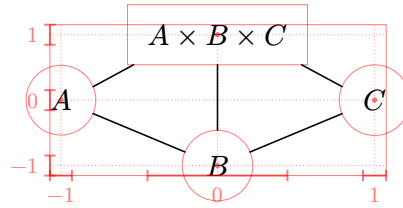
You can convert elastic coordinates to physical coordinates with a callback:



```
#import "@preview/cetz:0.1.2"
#arrow-diagram({
  let (A, B, C) = ((0,0), (1,1), (1,-1))
  node(A, $A$)
  node(B, $B$)
  node(C, $C$)
  conn(A, B, "hook->")
  conn(A, C, "->>")
  resolve-coords(A, B, callback: (p1, p2) => {
    cetz.draw.rect(
      (to: p1, rel: (-15pt, -15pt)),
      (to: p2, rel: (15pt, 15pt)),
      fill: rgb("00f1"),
      stroke: (paint: blue, dash: "dashed"),
    )
  })
})
```

Connectors

Lines between nodes connect to the node's bounding circle or bounding rectangle. The bounding shape is chosen automatically depending on the node's aspect ratio.



The defocus adjustment

For aesthetic reasons, a line connecting to a node should not necessarily be focused to the node's exact center, especially if the node is short and wide or tall and narrow. Notice how in the figure above the lines connecting to the node $A \times B \times C$ would intersect slightly above its center, making the diagram look more comfortable. The effect of this is shown below:

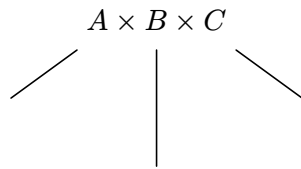


Figure 1: With defocus

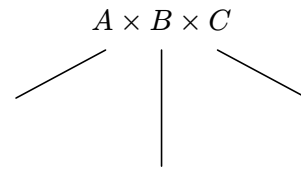
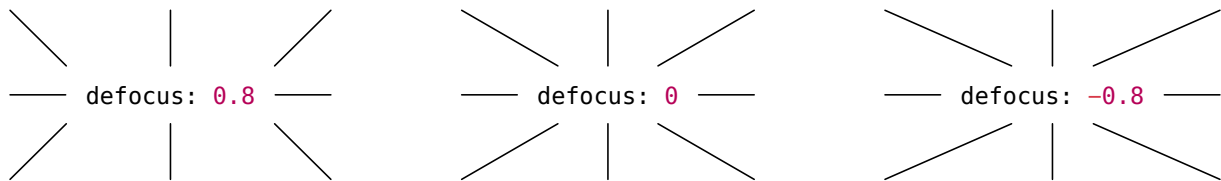


Figure 2: Without defocus

The amount is controlled by the defocus attribute of the diagram. It is best explained by example:



For defocus: 0, the connecting lines are directed exactly at the grid point at the node's center.

Function reference

arrow-diagram

Draw an arrow diagram.

Parameters

```
arrow-diagram(  
  ..args: array ,  
  gutter: length pair of lengths ,  
  debug: bool 1 2 3 ,  
  node-pad: length pair of lengths ,  
  cell-size: length pair of lengths ,  
  defocus: number  
)
```

..args array

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

gutter length or pair of lengths

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

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

Default: 3em

debug bool or 1 or 2 or 3

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

Default: false

node-pad length or pair of lengths

Padding between a node's content and its bounding box.

Default: 15pt

cell-size length or pair of lengths

Minimum size of all rows and columns.

Default: 0pt

defocus number

Strength of the defocus correction. 0 to disable.

Default: 0.2

conn

Draw a connector

Parameters

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

from elastic coord

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

to elastic coord

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

..args any

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

```
conn((0,0), (1,0), $f$, "->")  
conn((0,0), (1,0), $f$, marks: "->")  
conn((0,0), (1,0), "->", label: $f$)  
conn((0,0), (1,0), label: $f$, marks: "->")
```

label `content`

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

Default: `none`

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

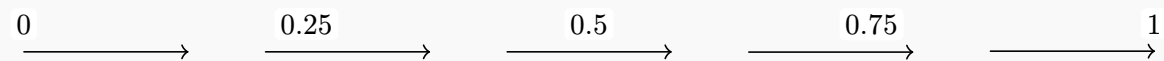
Which side of the connector to place the label on, viewed as you walk along it. If `center`, then the label is placed over the connector. When `auto`, a value of `left` or `right` is chosen to automatically so that the label is

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

Default: `auto`

label-pos `number`

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



Default: `0.5`

label-sep `number`

Separation between the connector and the label anchor.

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



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



Default: `0.4em`


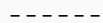
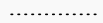

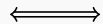
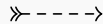
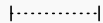
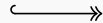
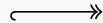

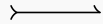
label-anchor `anchor`

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

Default: `auto`

marks pair of strings

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

Arrow	Shorthand	Arguments
	-	(marks: (none, none))
	--	(marks: (none, none), dash: "dashed")
	..	(marks: (none, none), dash: "dotted")
	->	(marks: (none, "head"))
	<=>	(marks: ("head", "head"), double: true)
	>>->	(marks: ("twotail", "head"), dash: "dashed")
	..	(marks: ("bar", "bar"), dash: "dotted")
	hook->	(marks: ("hook", "twohead"))
	hook'->	(marks: ("hook'", "twohead"))
	>-harpoon	(marks: ("tail", "harpoon"))
	>-harpoon'	(marks: ("tail", "harpoon'))

Default: (none, none)

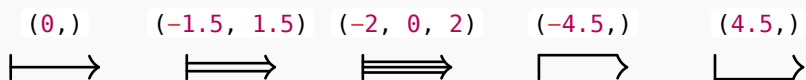
double bool

Shortcut for `extrude: (-1.5, 1.5)`, showing a double stroke.

Default: false

extrude array of numbers

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



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

Default: auto

crossing bool

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



Default: false

crossing-thickness number

Thickness of the white “crossing” background stroke, if `crossing: true`, in multiples of the normal stroke’s thickness.



Default: 5

resolve-coords

Resolve coordinates and pass them to a callback function

Parameters

```
resolve-coords(  
  ..args: point2f,  
  callback: function  
)
```

..args point2f

One or more dimensionless 2D points of the form (x, y).

callback function

Function to be called with the resolved coordinates as arguments.

Default: (`..args`) => `none`

compute-cells

Compute a lookup table of the attributes of each grid cell.

Parameters

```
compute-cells(  
  nodes: array,  
  grid: dictionary,  
  options  
)
```

nodes array

Array of nodes to consider when calculating the sizes of cells, where each node of the form:

```
(  
  pos: (i, j),  
  size: (width, height),  
)
```

grid dictionary

Grid specification of the form

```
(
  origin: (i, j),
  centers: ((x1, x2, ...), (y1, y2, ...)),
)
```

options

compute-grid

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

Parameters

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

nodes

options

expand-fractional-rects

Convert an array of rects with fractional positions into rects with integral positions.

If a rect is centered at a fractional position $\text{floor}(x) < x < \text{ceil}(x)$, it will be replaced by two new rects centered at $\text{floor}(x)$ and $\text{ceil}(x)$. The total width of the original rect is split across the two new rects according to which one is closer. (E.g., if the original rect is at $x = 0.25$, the new rect at $x = 0$ has 75% the original width and the rect at $x = 1$ has 25%.) The same splitting procedure is done for y positions and heights.

Parameters

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

rects array of rects

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

round-arrow-cap-offset

Calculate cap offset of round-style arrow cap

Parameters

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

r length

Radius of curvature of arrow cap.

θ angle

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

y length

Lateral offset from the central stroke line.

get-arc-connecting-points

Determine arc between two points with a given bend angle

The bend angle is the angle between chord of the arc (line connecting the points) and the tangent to the arc and the first point.

Parameters

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

from point

2D vector of initial point.

to point

2D vector of final point.

angle angle

The bend angle between chord of the arc (line connecting the points) and the tangent to the arc and the first point.

