The CeTZ Package

Johannes Wolf
fenjalien

Version 0.1.1

1 Introduction

This package provides a way to draw stuff using a similar API to Processing but with relative coordinates and anchors from TikZ. You also won't have to worry about accidentally drawing over other content as the canvas will automatically resize. And remember: up is positive!

The name CeTZ is a recursive acronym for "CeTZ, ein Typst Zeichenpacket" (german for "CeTZ, a Typst drawing package") and is pronounced like the word "Cats".

2 Usage

This is the minimal starting point:

```
#import "@local/cetz:0.2.0"
#cetz.canvas({
  import cetz.draw: *
  ...
})
```

Note that draw functions are imported inside the scope of the `canvas` block. This is recommended as draw functions override Typst's functions such as `line`.

2.1 Argument Types

Argument types in this document are formatted in `monospace` and encased in angle brackets <>. Types such as `<integer>` and `<content>` are the same as Typst but additional are required:

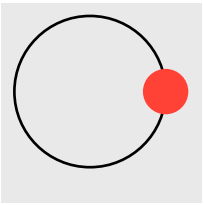
`<coordinate>` Any coordinate system. See Section 4.

`<number>` `<integer> or <float>`

2.2 Anchors

Anchors are named positions relative to named elements.

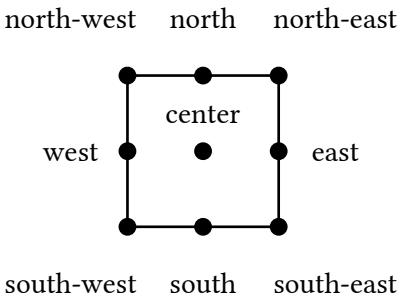
To use an anchor of an element, you must give the element a name using the `name` argument.



```
// Name the circle
circle((0,0), name: "circle")

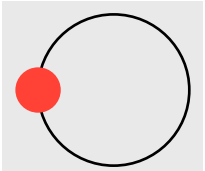
// Draw a smaller red circle at "circle"'s east anchor
fill(red)
stroke(none)
circle("circle.east", radius: 0.3)
```

Group elements will have default anchors based on their axis aligned bounding box, they are:



Other elements will have their own anchors.

Elements can be placed relative to their own anchors if they have an argument called `anchor:`:



```
// An element does not have to be named
// in order to use its own anchors.
circle((0,0), anchor: "west")

// Draw a smaller red circle at the origin
fill(red)
stroke(none)
circle((0,0), radius: 0.3)
```

3 Draw Function Reference

3.1 Canvas

```
canvas(background: none, length: 1cm, debug: false, body)
```

background `<color>` (default: none)
: A color to be used for the background of the canvas.

length `<length>` (default: 1cm)
: Used to specify what 1 coordinate unit is.

debug `<bool>` (default: false)
: Shows the bounding boxes of each element when `true`.

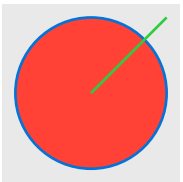
body
: A code block in which functions from `draw.typ` have been called.

3.2 Styling

You can style draw elements by passing the relevant named arguments to their draw functions. All elements have stroke and fill styling unless said otherwise.

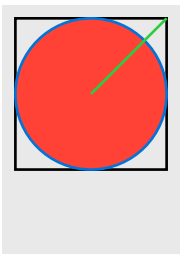
fill `<color>` or `<none>` (default: none)
: How to fill the draw element.

stroke `<none>` or `<auto>` or `<length>` (default: black + 1pt)
or `<color>` or `<dictionary>` or `<stroke>`
: How to stroke the border or the path of the draw element. See Typst's line documentation for more details: https://typst.app/docs/reference/visualize/line/#parameters-stroke



```
// Draws a red circle with a blue border
circle((0, 0), fill: red, stroke: blue)
// Draws a green line
line((0, 0), (1, 1), stroke: green)
```

Instead of having to specify the same styling for each time you want to draw an element, you can use the `set-style` function to change the style for all elements after it. You can still pass styling to a draw function to override what has been set with `set-style`. You can also use the `fill()` and `stroke()` functions as a shorthand to set the fill and stroke respectively.



```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

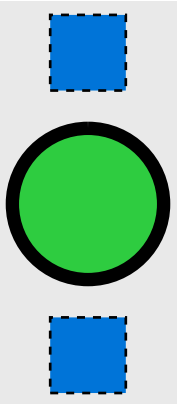
// Sets the global style to have a fill of red and a stroke of blue
set-style(stroke: blue, fill: red)
circle((0,0))

// Draws a green line despite the global stroke is blue
line((), (1,1), stroke: green)
```

When using a dictionary for a style, it is important to note that they update each other instead of overriding the entire option like a non-dictionary value would do. For example, if the stroke is set to `(paint: red, thickness: 5pt)` and you pass `(paint: blue)`, the stroke would become `(paint: blue, thickness: 5pt)`.

```
// Sets the stroke to red with a thickness of 5pt
set-style(stroke: (paint: red, thickness: 5pt))
// Draws a line with the global stroke
line((0,0), (1,0))
// Draws a blue line with a thickness of 5pt because dictionaries update the style
line((0,0), (1,1), stroke: (paint: blue))
// Draws a yellow line with a thickness of 1pt because other values override the style
line((0,0), (0,1), stroke: yellow)
```

You can also specify styling for each type of element. Note that dictionary values will still update with its global value, the full hierarchy is `function > element type > global`. When the value of a style is `auto`, it will become exactly its parent style.



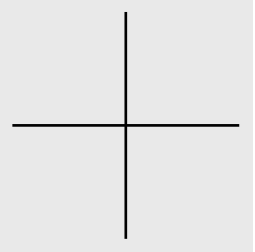
```
set-style(
  // Global fill and stroke
  fill: green,
  stroke: (thickness: 5pt),
  // Stroke and fill for only rectangles
  rect: (stroke: (dash: "dashed"), fill: blue),
)
rect((0,0), (1,1))
circle((0.5, -1.5))
rect((0,-3), (1, -4), stroke: (thickness: 1pt))
```

```
// Its a nice drawing okay
set-style(
  rect: (
    fill: red,
    stroke: none
  ),
  line: (
    fill: blue,
    stroke: (dash: "dashed")
  ),
)
rect((0,0), (1,1))

line((0, -1.5), (0.5, -0.5), (1, -1.5), close: true)

circle((0.5, -2.5), radius: 0.5, fill: green)
```

3.3 Elements

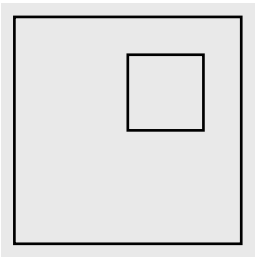


```
line((-1.5, 0), (1.5, 0))
line((0, -1.5), (0, 1.5))
```

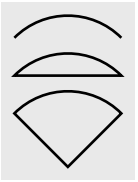
Styling

mark `<dictionary> or <auto>` (default: `auto`)

The styling to apply to marks on the line, see `mark`



```
rect((0,0), (1,1))
rect((-1.5, 1.5), (1.5, -1.5))
```



```
arc((0,0), start: 45deg, stop: 135deg)
arc((0,-0.5), start: 45deg, delta: 90deg, mode: "CLOSE")
arc((0,-1), stop: 135deg, delta: 90deg, mode: "PIE")
```

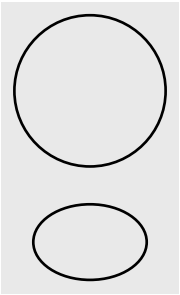
Styling

radius `<number> or <array>` (default: 1)

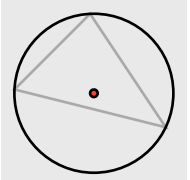
The radius of the arc. This is also a global style shared with circle!

mode `<string>` (default: `"OPEN"`)

The options are “OPEN” (the default, just the arc), “CLOSE” (a circular segment) and “PIE” (a circular sector).



```
circle((0,0))
// Draws an ellipse
circle((0,-2), radius: (0.75, 0.5))
```

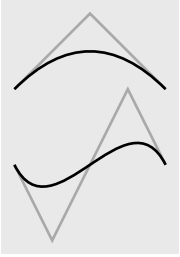


```
let (a, b, c) = ((0,0), (2,-.5), (1,1))
line(a, b, c, close: true, stroke: gray)
circle-through(a, b, c, name: "c")
circle("c.center", radius: .05, fill: red)
```

Styling

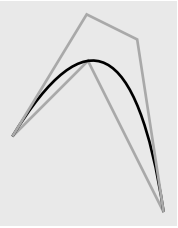
radius `<number> or <length> or <array of <number> or <length>>` (default: 1)

The circle’s radius. If an array is given an ellipse will be drawn where the first item is the `x` radius and the second item is the `y` radius. This is also a global style shared with arc!



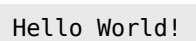
```
let (a, b, c) = ((0, 0), (2, 0), (1, 1))
line(a, c,  b, stroke: gray)
bezier(a, b, c)

let (a, b, c, d) = ((0, -1), (2, -1), (.5, -2), (1.5, 0))
line(a, c, d, b, stroke: gray)
bezier(a, b, c, d)
```

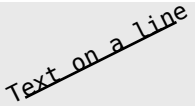


```
let (a, b, c) = ((0, 0), (1, 1), (2, -1))
line(a, b, c, stroke: gray)
bezier-through(a, b, c, name: "b")

// Show calculated control points
line(a, "b.ctrl-0", "b.ctrl-1", c, stroke: gray)
```

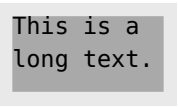


```
content((0,0), [Hello World!])
```



```
let (a, b) = ((1,0), (3,1))

line(a, b)
content((a, .5, b), angle: b, [Text on a line], anchor: "south")
```

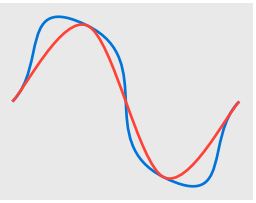


```
content((0,0), (2,1), par(justify: false)[This is a long text.], frame: "rect",
  fill: gray, stroke: none)
```

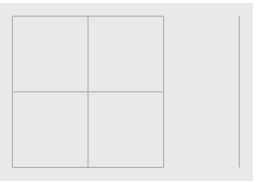
Styling

This draw element is not affected by fill or stroke styling.

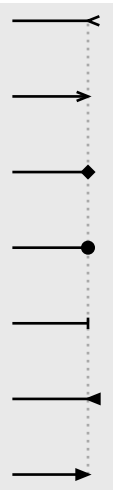
padding `<length>` (default: `0pt`)



```
catmull((0,0), (1,1), (2,-1), (3,0), tension: .4, stroke: blue)
catmull((0,0), (1,1), (2,-1), (3,0), tension: .5, stroke: red)
```



```
grid((0,0), (3,2), help-lines: true)
```



```
line((1, 0), (1, 6), stroke: (paint: gray, dash: "dotted"))
set-style(mark: (fill: none))
line((0, 6), (1, 6), mark: (end: "<"))
line((0, 5), (1, 5), mark: (end: ">"))
set-style(mark: (fill: black))
line((0, 4), (1, 4), mark: (end: "<>"))
line((0, 3), (1, 3), mark: (end: "o"))
line((0, 2), (1, 2), mark: (end: "|"))
line((0, 1), (1, 1), mark: (end: "<"))
line((0, 0), (1, 0), mark: (end: ">"))
```

Styling

symbol `<string>` (default: >)

The type of mark to draw when using the `mark` function.

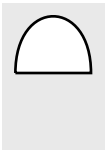
start `<string>`
The type of mark to draw at the start of a path.

end `<string>`
The type of mark to draw at the end of a path.

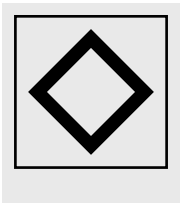
size `<number>` (default: 0.15)
The size of the marks.

angle `<angle>` (default: `45deg`)
Angle for triangle style marks ("<" and ">")

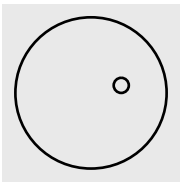
3.4 Path Transformations



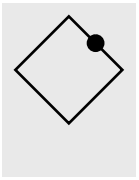
```
// Merge two different paths into one
merge-path({
  line((0, 0), (1, 0))
  bezier((), (0, 0), (1,1), (0,1))
}, fill: white)
```



```
// Create group
group({
  stroke(5pt)
  scale(.5); rotate(45deg)
  rect((-1,-1),(1,1))
})
rect((-1,-1),(1,1))
```

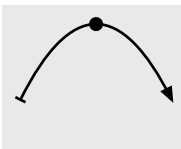


```
group(name: "g", {
  circle((0,0))
  anchor("x", (.4,.1))
})
circle("g.x", radius: .1)
```

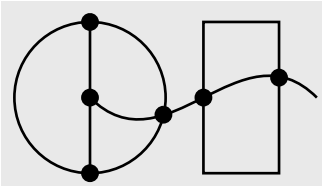


```
group(name: "g", {
  rotate(45deg)
  rect((0,0), (1,1), name: "r")
  copy-anchors("r")
})
circle("g.north", radius: .1, fill: black)
```

```
place-anchors(name: "demo",
              bezier((0,0), (3,0), (1,-1), (2,1)),
              (name: "a", pos: .15),
              (name: "mid", pos: .5))
circle("demo.a", radius: .1, fill: black)
circle("demo.mid", radius: .1, fill: black)
```



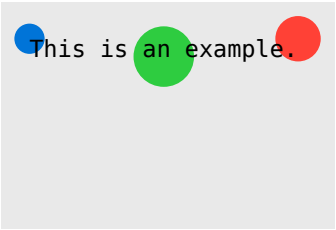
```
place-marks(bezier-through((0,0), (1,1), (2,0)),
            (mark: "|", size: .1, pos: 0),
            (mark: "o", size: .2, pos: .5),
            (mark: ">", size: .3, pos: 1),
            fill: black)
```



```
intersections("demo", {
  circle((0, 0))
  bezier((0,0), (3,0), (1,-1), (2,1))
  line((0,-1), (0,1))
  rect((1.5,-1),(2.5,1))
})
for-each-anchor("demo", (name) => {
  circle("demo." + name, radius: .1, fill: black)
})
```

3.5 Layers

You can use layers to draw elements below or on top of other elements by using layers with a higher or lower index. When rendering, all draw commands are sorted by their layer (0 being the default).



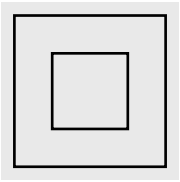

```
// Draw something behind text
set-style(stroke: none)
content((0, 0), [This is an example.], name: "text")
on-layer(-1, {
  circle("text.north-east", radius: .3, fill: red)
  circle("text.south", radius: .4, fill: green)
  circle("text.north-west", radius: .2, fill: blue)
})
```

3.6 Transformations

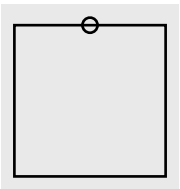
All transformation functions push a transformation matrix onto the current transform stack. To apply transformations scoped use a `group(...)` object.

Transformation martices get multiplied in the following order:

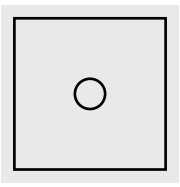
$$M_{\text{world}} = M_{\text{world}} \cdot M_{\text{local}}$$



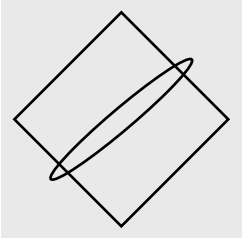
```
// Outer rect
rect((0,0), (2,2))
// Inner rect
translate((.5,.5,0))
rect((0,0), (1,1))
```



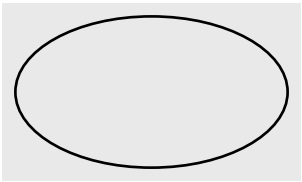
```
// Outer rect
rect((0,0), (2,2), name: "r")
// Move origin to top edge
set-origin("r.north")
circle((0, 0), radius: .1)
```



```
rect((0,0), (2,2))
set-viewport((0,0), (2,2), bounds: (10, 10))
circle((5,5))
```



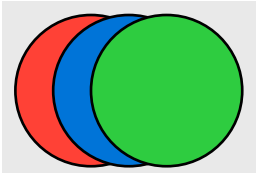
```
// Rotate on z-axis
rotate((z: 45deg))
rect((-1,-1), (1,1))
// Rotate on y-axis
rotate((y: 80deg))
circle((0,0))
```



```
// Scale x-axis
scale((x: 1.8))
circle((0,0))
```

3.7 Context Modification

The context of a canvas holds the canvas' internal state like style and transformation. Note that the fields of the context of a canvas are considered private and therefore unstable. You can add custom values to the context, but in order to prevent naming conflicts with future CeTZ versions, try to assign unique names.



```
// Setting a custom transformation matrix
set-ctx(ctx => {
  let mat = ((1, 0, .5, 0),
             (0, 1, 0, 0),
             (0, 0, 1, 0),
             (0, 0, 0, 1))
  ctx.transform = mat
  return ctx
})
circle((z: 0), fill: red)
circle((z: 1), fill: blue)
circle((z: 2), fill: green)
```

```
(
  (1, 0, 0.5, 0),
  (0, −1, −0.5, 0),
  (0, 0, 1, 0),
  (0, 0, 0, 1),
)
```

```
// Print the transformation matrix
get-ctx(ctx => {
  content((), [#repr(ctx.transform)])
})
```

4 Coordinate Systems

A *coordinate* is a position on the canvas on which the picture is drawn. They take the form of dictionaries and the following sub-sections define the key value pairs for each system. Some systems have a more implicit form as an array of values and `CeTZ` attempts to infer the system based on the element types.

4.1 XYZ

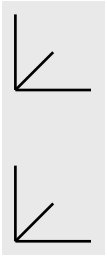
Defines a point `x` units right, `y` units upward, and `z` units away.

x `<number>` or `<length>` (default: 0)
The number of units in the `x` direction.

y `<number>` or `<length>` (default: 0)
The number of units in the `y` direction.

z `<number>` or `<length>` (default: 0)
The number of units in the `z` direction.

The implicit form can be given as an array of two or three `<number>` or `<length>`, as in `(x,y)` and `(x,y,z)`.

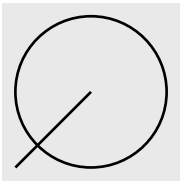


```
line((0,0), (x: 1))
line((0,0), (y: 1))
line((0,0), (z: 1))

// Implicit form
line((0, -2), (1, -2))
line((0, -2), (0, -1, 0))
line((0, -2), (0, -2, 1))
```

4.2 Previous

Use this to reference the position of the previous coordinate passed to a draw function. This will never reference the position of a coordinate used in to define another coordinate. It takes the form of an empty array `()`. The previous position initially will be `(0, 0, 0)`.



```
line((0,0), (1, 1))

// Draws a circle at (1,1)
circle(())
```

4.3 Relative

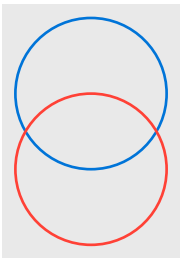
Places the given coordinate relative to the previous coordinate. Or in other words, for the given coordinate, the previous coordinate will be used as the origin. Another coordinate can be given to act as the previous coordinate instead.

rel `<coordinate>`
: The coordinate to be place relative to the previous coordinate.

update `<bool>` (default: `true`)
: When false the previous position will not be updated.

to `<coordinate>` (default: `()`)
: The coordinate to treat as the previous coordinate.

In the example below, the red circle is placed one unit below the blue circle. If the blue circle was to be moved to a different position, the red circle will move with the blue circle to stay one unit below.



```
circle((0, 0), stroke: blue)
circle((rel: (0, -1)), stroke: red)
```

4.4 Polar

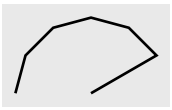
Defines a point a `radius` distance away from the origin at the given `angle`.

angle `<angle>`
: The angle of the coordinate. An angle of `0deg` is to the right, a degree of `90deg` is upward. See https://typst.app/docs/reference/layout/angle/ for details.

radius `<number> or <length> or <array of length or number>`
: The distance from the origin. An array can be given, in the form `(x, y)` to define the `x` and `y` radii of an ellipse instead of a circle.

```
line((0,0), (angle: 30deg, radius: 1cm))
```

The implicit form is an array of the angle then the radius `(angle, radius)` or `(angle, (x, y))`.



```
line((0,0), (30deg, 1), (60deg, 1),
     (90deg, 1), (120deg, 1), (150deg, 1), (180deg, 1))
```

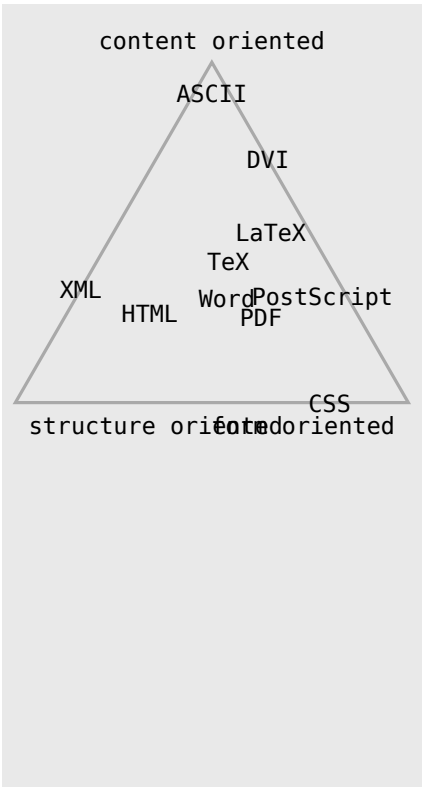
4.5 Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2 ..., v_n and numbers α_1, α_2, ..., α_n. Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_1 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

bary `<dictionary>`

A dictionary where the key is a named element and the value is a `<float>`. The `center` anchor of the named element is used as v and the value is used as a.




```
circle((90deg, 3), radius: 0, name: "content")
circle((210deg, 3), radius: 0, name: "structure")
circle((-30deg, 3), radius: 0, name: "form")

for (c, a) in (
  ("content", "south"),
  ("structure", "north-west"),
  ("form", "north-east")
) {
  content(c, box(c + " oriented", inset: 5pt), anchor: a)
}

stroke(gray + 1.2pt)
line("content", "structure", "form", close: true)

for (c, s, f, cont) in (
  (0.5, 0.1, 1, "PostScript"),
  (1, 0, 0.4, "DVI"),
  (0.5, 0.5, 1, "PDF"),
  (0, 0.25, 1, "CSS"),
  (0.5, 1, 0, "XML"),
  (0.5, 1, 0.4, "HTML"),
  (1, 0.2, 0.8, "LaTeX"),
  (1, 0.6, 0.8, "TeX"),
  (0.8, 0.8, 1, "Word"),
  (1, 0.05, 0.05, "ASCII")
) {
  content((bary: (content: c, structure: s, form: f)), cont)
}
```

4.6 Anchor

Defines a point relative to a named element using anchors, see Section 2.2.

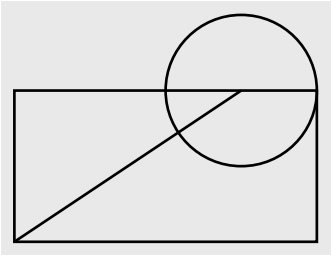
name `<string>`

The name of the element that you wish to use to specify a coordinate.

anchor `<string>`

An anchor of the element. If one is not given a default anchor will be used. On most elements this is `center` but it can be different.

You can also use implicit syntax of a dot separated string in the form `"name.anchor"`.



```
line((0,0), (3,2), name: "line")
circle("line.end", name: "circle")
rect("line.start", "circle.east")
```

4.7 Tangent

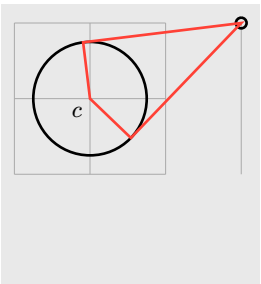
This system allows you to compute the point that lies tangent to a shape. In detail, consider an element and a point. Now draw a straight line from the point so that it "touches" the element (more formally, so that it is *tangent* to this element). The point where the line touches the shape is the point referred to by this coordinate system.

element `<string>`
: The name of the element on whose border the tangent should lie.

point `<coordinate>`
: The point through which the tangent should go.

solution `<integer>`
: Which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently it does this by assuming the distance between the center and top anchor (See Section 2.2) is the radius of a circle.




```
grid((0,0), (3,2), help-lines: true)

circle((3,2), name: "a", radius: 2pt)
circle((1,1), name: "c", radius: 0.75)
content("c", $ c $, anchor: "north-east", padding: .1)

stroke(red)
line("a", (element: "c", point: "a", solution: 1),
     "c", (element: "c", point: "a", solution: 2),
     close: true)
```

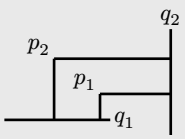
4.8 Perpendicular

Can be used to find the intersection of a vertical line going through a point p and a horizontal line going through some other point q.

horizontal `<coordinate>`
: The coordinate through which the horizontal line passes.

vertical `<coordinate>`
: The coordinate through which the vertical line passes.

You can use the implicit syntax of `(horizontal, "-|", vertical)` or `(vertical, "|-", horizontal)`




```
set-style(content: (padding: .05))
content((30deg, 1), $ p_1 $, name: "p1")
content((75deg, 1), $ p_2 $, name: "p2")

line((-0.2, 0), (1.2, 0), name: "xline")
content("xline.end", $ q_1 $, anchor: "west")
line((2, -0.2), (2, 1.2), name: "yline")
content("yline.end", $ q_2 $, anchor: "south")

line("p1.south-east", (horizontal: (), vertical: "xline.end"))
line("p2.south-east", ((), "|-", "xline.end")) // Short form
line("p1.south-east", (vertical: (), horizontal: "yline.end"))
line("p2.south-east", ((), "-|", "yline.end")) // Short form
```

4.9 Interpolation

Use this to linearly interpolate between two coordinates `a` and `b` with a given factor `number`. If `number` is a `<length>` the position will be at the given distance away from `a` towards `b`. An angle can also be given for the general meaning: "First consider the line from `a` to `b`. Then rotate this line by `angle` around point `a`. Then the two endpoints of this line will be `a` and some point `c`. Use this point `c` for the subsequent computation."

a `<coordinate>`
The coordinate to interpolate from.

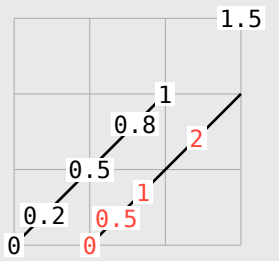
b `<coordinate>`
The coordinate to interpolate to.

number `<number>` or `<length>`
The factor to interpolate by or the distance away from `a` towards `b`.

angle `<angle>` (default: `0deg`)

abs `<bool>` (default: `false`)
Interpret `number` as absolute distance, instead of a factor.

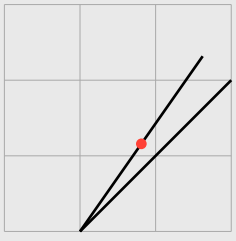
Can be used implicitly as an array in the form `(a, number, b)` or `(a, number, angle, b)`.



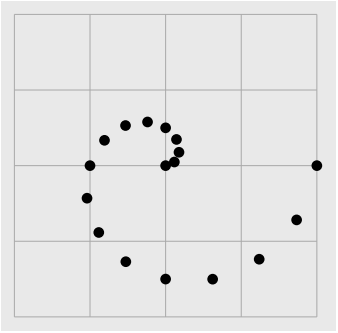

```
grid((0,0), (3,3), help-lines: true)

line((0,0), (2,2))
for i in (0, 0.2, 0.5, 0.8, 1, 1.5) { /* Relative distance */
  content(((0,0), i, (2,2)),
          box(fill: white, inset: 1pt, [#i]))
}

line((1,0), (3,2))
for i in (0, 0.5, 1, 2) { /* Absolute distance */
  content((a: (1,0), number: i, abs: true, b: (3,2)),
          box(fill: white, inset: 1pt, text(red, [#i])))
}
```



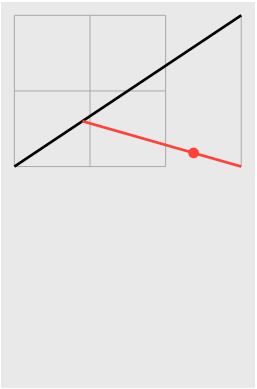
```
grid((0,0), (3,3), help-lines: true)
line((1,0), (3,2))
line((1,0), ((1, 0), 1, 10deg, (3,2)))
fill(red)
stroke(none)
circle(((1, 0), 0.5, 10deg, (3, 2)), radius: 2pt)
```



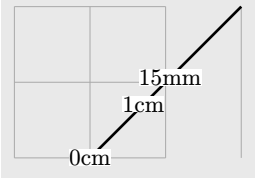
```
grid((0,0), (4,4), help-lines: true)

fill(black)
stroke(none)
let n = 16
for i in range(0, n+1) {
  circle(((2,2), i / 8, i * 22.5deg, (3,2)), radius: 2pt)
}
```

You can even chain them together!



```
grid((0,0), (3, 2), help-lines: true)
line((0,0), (3,2))
stroke(red)
line(((0,0), 0.3, (3,2)), (3,0))
fill(red)
stroke(none)
circle(
  ( // a
    (((0, 0), 0.3, (3, 2))),
    0.7,
    (3,0)
  ),
  radius: 2pt
)
```

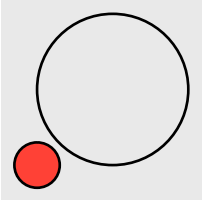



```
grid((0,0), (3, 2), help-lines: true)
line((1,0), (3,2))
for (l, c) in ((0cm, "0cm"), (1cm, "1cm"), (15mm, "15mm")) {
  content(((1,0), l, (3,2)), box(fill: white, $ #c $))
}
```

4.10 Function

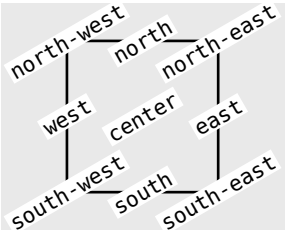
An array where the first element is a function and the rest are coordinates will cause the function to be called with the resolved coordinates. The resolved coordinates have the same format as the implicit form of the 3-D XYZ coordinate system, Section 4.1.

The example below shows how to use this system to create an offset from an anchor, however this could easily be replaced with a relative coordinate with the `to` argument set, Section 4.3.



```
circle((0, 0), name: "c")
fill(red)
circle((v => cetz.vector.add(v, (0, -1)), "c.west"), radius: 0.3)
```

5 Utility




```
// Label nodes anchors
rect((0, 0), (2,2), name: "my-rect")
for-each-anchor("my-rect", (name) => {
  content((), box(inset: 1pt, fill: white, text(8pt, [#name])),
          angle: -30deg)
})
```

6 Libraries

6.1 Tree

With the tree library, CeTZ provides a simple tree layout algorithm.

6.1.1 tree

Layout and render tree nodes

Parameters

```
tree(
  root: array,
  draw-node: function,
  draw-edge: function,
  direction: string,
  parent-position: string,
  grow: float,
  spread: float,
  name,
  ..style
)
```

root `array`

Tree structure represented by nested lists Example: `([root], [child 1], ([child 2], [grandchild 1]))`

draw-node `function`

Callback for rendering a node. Signature: `(node) => elements`. The nodes position is accessible through the anchor `"center"` or the last position `()`.

Default: `auto`

draw-edge `function`

Callback for rendering edges between nodes Signature: `(source-name, target-name, target-node) => elements`

Default: `auto`

direction `string`

Tree grow direction (up, down, left, right)

Default: `"down"`

parent-position `string`

Positioning of parent nodes (begin, center, end)

Default: `"center"`

grow `float`

Depth grow factor (default 1)

Default: 1

spread `float`

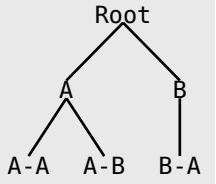
Sibling spread factor (default 1)

Default: 1

name

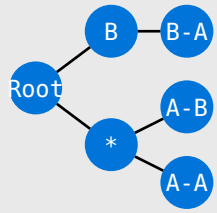
Default: none

..style




```
import cetz.tree

let data = ([Root], ([A], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), line: (stroke: blue))
```




```
import cetz.tree

let data = ([Root], ([\*], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), direction: "right",
          mark: (end: ">", fill: none),
          draw-node: (node, ..) => {
            circle((), radius: .35, fill: blue, stroke: none)
            content((), text(white, [#node.content]))
          },
          draw-edge: (from, to, ..) => {
            let (a, b) = (from + ".center",
                          to + ".center")

             line((a: a, b: b, abs: true, number: .35),
                  (a: b, b: a, abs: true, number: .35))
          })
```

6.1.2 Node

A tree node is an array of nodes. The first array item represents the current node, all following items are direct children of that node. The node itselfes can be of type `content` or `dictionary` with a key `content`.

6.2 Plot

The library `plot` of CeTZ allows plotting 2D data.

- add-anchor()
- plot()

6.2.1 add-anchor

Add an anchor to a plot environment

Parameters

```
add-anchor(
  name: string,
  position: array,
  axes: array
)
```

name `string`

Anchor name

position `array`

Tuple of x and y values. Both values can have the special values “min” and “max”, which resolve to the axis min/max value. Position is in axis space!

axes `array`

Name of the axes to use (“x”, “y”), note that both axes must exist!

Default: `("x", "y")`

6.2.2 plot

Create a plot environment

Note: Data for plotting must be passed via `plot.add(..)`

Note that different axis-styles can show different axes. The `"school-book"` and `"left"` style shows only axis “x” and “y”, while the `"scientific"` style can show “x2” and “y2”, if set (if unset, “x2” mirrors “x” and “y2” mirrors “y”). Other axes (e.G. “my-axis”) work, but no ticks or labels will be shown.

Options

The following options are supported per axis and must be prefixed by `<axis-name>-`, e.G. `x-min: 0` or `y-label: [y]`.

- label (content): Axis label
- min (int): Axis minimum value
- max (int): Axis maximum value
- tick-step (none, float): Distance between major ticks (or no ticks if none)
- minor-tick-step (none, float): Distance between minor ticks (or no ticks if none)
- ticks (array): List of ticks values or value/label tuples. Example `(1,2,3)` or `((1, [A]), (2, [B]),)`
- format (string): Tick label format, `"float"`, `"sci"` (scientific) or a custom function that receives a value and returns a content (`value => content`).
- grid (bool,string): Enable grid-lines at tick values:
  - `"major"`: Enable major tick grid
  - `"minor"`: Enable minor tick grid
  - `"both"`: Enable major & minor tick grid
  - `false`: Disable grid
- unit (none, content): Tick label suffix
- decimals (int): Number of decimals digits to display for tick labels

Parameters

```
plot(
  body: body,
  size: array,
  axis-style: string,
  name: string,
  plot-style: style function,
  mark-style: style function,
  fill-below: bool,
  ..options: any
)
```

body `body`

Calls of `plot.add` or `plot.add-*` commands

size `array`

Plot canvas size tuple of width and height in canvas units

Default: `(1, 1)`

axis-style `string`

Axis style “scientific”, “left”, “school-book”

- `"scientific"`: Frame plot area and draw axes y, x, y2, and x2 around it
- `"school-book"`: Draw axes x and y as arrows with both crossing at $(0, 0)$
- `"left"`: Draw axes x and y as arrows, the y axis stays on the left (at `x.min`) and the x axis at the bottom (at `y.min`)

Default: `"scientific"`

name `string`

Element name

Default: `none`

plot-style `style` or `function`

Style used for drawing plot graphs This style gets inherited by all plots.

Default: `default-plot-style`

mark-style `style` or `function`

Style used for drawing plot marks. This style gets inherited by all plots.

Default: `default-mark-style`

fill-below `bool`

Fill functions below the axes (draw axes above fills)

Default: `true`

..options `any`

The following options are supported per axis and must be prefixed by `<axis-name>-`, e.G. `x-min: 0`.

- min (int): Axis minimum
- max (int): Axis maximum
- tick-step (float): Major tick step
- minor-tick-step (float): Major tick step
- ticks (array): List of ticks values or value/label tuples
- unit (content): Tick label suffix
- decimals (int): Number of decimals digits to display

- add()
- add-hline()
- add-vline()

6.2.3 add

Add data to a plot environment.

Note: You can use this for scatter plots by setting the stroke style to `none`: `add(..., style: (stroke: none))`.

Must be called from the body of a `plot(..)` command.

Parameters

```
add(
  domain: array,
  hypograph: bool,
  epigraph: bool,
  fill: bool,
  fill-type: string,
  style: style,
  mark: string,
  mark-size: float,
  mark-style,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: array,
  data: array function
)
```

domain `array`

Domain tuple of the plot. If `data` is a function, domain must be specified, as `data` is sampled for x-values in `domain`. Values must be numbers.

Default: `auto`

hypograph `bool`

Fill hypograph; uses the `hypograph` style key for drawing

Default: `false`

epigraph `bool`

Fill epigraph; uses the `epigraph` style key for drawing

Default: `false`

fill `bool`

Fill to y zero

Default: `false`

fill-type `string`

Fill type:

- **`"axis"`** Fill to y = 0
- **`"shape"`** Fill the functions shape

Default: `"axis"`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

mark `string`

Mark symbol to place at each distinct value of the graph. Uses the `mark` style key of `style` for drawing.

The following marks are supported:
- `"*"` or `"x"` – X
- `"+"` – Cross
- `"|"` – Bar
- `"-"` – Dash
- `"o"` – Circle
- `"triangle"` – Triangle
- `"square"` – Square

Default: `none`

mark-size `float`

Mark size in cavas units

Default: `.2`

mark-style

Default: `(:)`

samples `int`

Number of times the `data` function gets called for sampling y-values. Only used if `data` is of type function.

Default: `50`

sample-at `array`

Array of x-values the function gets sampled at in addition to the default sampling.

Default: `()`

line `string` or `dictionary`

Line type to use. The following types are supported:
- **`"linear"`** Linear line segments
- **`"spline"`** A smoothed line
- **`"vh"`** Move vertical and then horizontal
- **`"hv"`** Move horizontal and then vertical
- **`"vhv"`** Add a vertical step in the middle
- **`"raw"`** Like linear, but without linearization.

`"linear"` *should* never look different than `"raw"`.

If the value is a dictionary, the type must be supplied via the `type` key. The following extra attributes are supported:
- **`"samples" <int>`** Samples of splines
- **`"tension" <float>`** Tension of splines
- **`"mid" <float>`** Mid-Point of vhv lines (0 to 1)
- **`"epsilon" <float>`** Linearization slope epsilon for use with `"linear"`, defaults to 0.

Default: `"linear"`

axes `array`

Name of the axes to use ("x", "y"), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

data `array` or `function`

Array of 2D data points (numeric) or a function of the form `x => y`, where `x` is a value insides `domain` and `y` must be numeric or a 2D vector (for parametric functions).

Examples
- `((0,0), (1,1), (2,-1))`
- x => calc.pow(x, 2)

6.2.4 add-hline

Add horizontal lines at values y

Parameters

add-hline(
  ..y: `number`,
  axes: `array`,
  style: `style`
)

..y `number`

Y axis value(s) to add a line at

axes `array`

Name of the axes to use (“x”, “y”), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

6.2.5 add-vline

Add vertical lines at values x.

Parameters

```
add-vline(
  ..x: number,
  axes: array,
  style: style
)
```

..x `number`

X axis values to add a line at

axes `array`

Name of the axes to use (“x”, “y”), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

- add-contour()

6.2.6 add-contour

Add a contour plot of a sampled function or a matrix.

Parameters

```
add-contour(
  data: array function,
  z: float array,
  x-domain: array,
  y-domain: array,
  x-samples: int,
  y-samples: int,
  interpolate: bool,
  op: auto string function,
  axes: array,
  style: style,
  fill: bool,
  limit: int
)
```

data `array` or `function`

A function of the signature `(x, y) => z` or an array of floats where the first index is the row and the second index is the column.

Examples:

- `(x, y) => x > 0`
- `(x, y) => 30 - (calc.pow(1 - x, 2)+calc.pow(1 - y, 2))`

z `float` or `array`

Z values to plot. Contours containing values above z (z >= 0) or below z (z < 0) get plotted. If you specify multiple z values, they get plotted in order.

Default: `(1,)`

x-domain `array`

X axis domain tuple (min, max)

Default: `(0, 1)`

y-domain `array`

Y axis domain tuple (min, max)

Default: `(0, 1)`

x-samples `int`

X axis domain samples (2 < n)

Default: `25`

y-samples `int`

Y axis domain samples (2 < n)

Default: `25`

interpolate `bool`

Use linear interpolation between sample values

Default: `true`

op `auto` or `string` or `function`

Z value comparison oparator:

`">", ">=", "<", "<=", "!=", "=="` Use the operator for comparison.

`auto` Use ">=" for positive z values, "<=" for negative z values.

`<function>` Call comparison function of the format `(plot-z, data-z) => boolean`, where `plot-z` is the z-value from the plots `z` argument and `data-z` is the z-value of the data getting plotted.

Default: `auto`

axes `array`

Name of the axes to use ("x", "y"), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

fill `bool`

Fill each contour

Default: `false`

limit `int`

Limit of contours to create per z value before the function panics

Default: `50`

- add-boxwhisker()

6.2.7 add-boxwhisker

Add one or more box or whisker plots

Parameters

add-boxwhisker(
  data: `array` `dictionary`,
  axes: `array`,
  style: `style`,
  box-width: `float`,
  whisker-width: `float`,
  mark: `string`,
  mark-size: `float`
)

data `array` or `dictionary`

dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

The following fields are supported:

- `x` (number) X-axis value
- `min` (number) Minimum value
- `max` (number) Maximum value
- `q1`, `q2`, `q3` (number) Quartiles from lower to to upper
- `outliers` (array of numbers) Optional outliers

Examples:

```
• (x: 1                    // Location on x-axis
   outliers: (7, 65, 69), // Optional outlier values
   min: 15, max: 60        // Minimum and maximum
   q1: 25,                 // Quartiles: Lower
   q2: 35,                 //            Median
   q3: 50)                 //            Upper
```

axes `array`

Name of the axes to use (“x”, “y”), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

box-width `float`

Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75

Default: `0.75`

whisker-width `float`

Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5

Default: `0.5`

mark `string`

Mark to use for plotting outliers. Set `none` to disable. Defaults to “x”

Default: `"*"`

mark-size `float`

Size of marks for plotting outliers. Defaults to 0.15

Default: `0.15`

6.2.8 sample-fn

Sample the given one parameter function with `samples` values evenly spaced within the range given by `domain` and return each sampled `y` value in an array as `(x, y)` tuple.

If the functions first return value is a tuple `(x, y)`, then all return values must be a tuple.

Parameters

```
sample-fn(
  fn: function,
  domain: array,
  samples: int,
  sample-at: array
) -> array: Array of (x y) tuples
```

fn `function`

Function to sample of the form `(x) => y` or `(x) => (x, y)`.

domain `array`

X domain tuple (min, max), that is the minimum and maximum x value the function gets sampled at.

samples `int`

Number of samples in domain.

sample-at `array`

List of x values the function gets sampled at in addition to the `samples` number of samples. Values outsides the specified domain are legal.

Default: `()`

6.2.9 sample-fn2

Samples the given two parameter function with `x-samples` and `y-samples` values evenly spaced within the range given by `x-domain` and `y-domain` and returns each sampled output in an array.

Parameters

```
sample-fn2(
  fn: function,
  x-domain: array,
  y-domain: array,
  x-samples: int,
  y-samples: int
) -> array: Array of z scalars
```

fn `function`

Function of the form `(x, y) => z` with all values being numbers.

x-domain `array`

X domain tuple (min, max), that is the range of x values the function gets sampled between.

y-domain `array`

Y domain tuple (min, max), that is the range of y values the function gets sampled between.

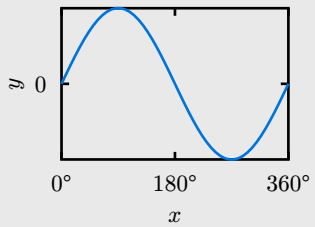
x-samples `int`

Number of samples in the x-domain.

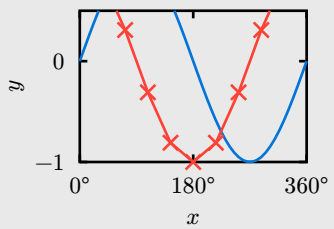
y-samples `int`

Number of samples in the y-domain.

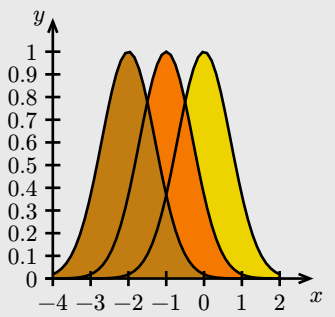
6.2.10 Examples




```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, {
  plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
})
```



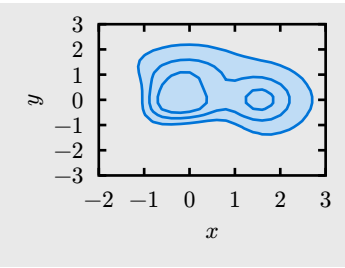

```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, y-max: .5, {
  plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
  plot.add(domain: (0, 360), x => calc.cos(x * 1deg),
           samples: 10, mark: "x", style: (mark: (stroke: blue)))
})
```




```
import cetz.plot
import cetz.palette

// Axes can be styled!
// Set the tick length to .1:
set-style(axes: (tick: (length: .1)))

// Plot something
plot.plot(size: (3,3), x-tick-step: 1, axis-style: "left", {
  for i in range(0, 3) {
    plot.add(domain: (-4, 2),
      x => calc.exp(-(calc.pow(x + i, 2))),
      fill: true, style: palette.tango)
  }
})
```




```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 1, y-tick-step: 1, {
  let z(x, y) = {
    (1 - x/2 + calc.pow(x,5) + calc.pow(y,3)) * calc.exp(-(x*x) - (y*y))
  }
  plot.add-contour(x-domain: (-2, 3), y-domain: (-3, 3),
                   z, z: (.1, .4, .7), fill: true)
})
```

6.2.11 Styling

The following style keys can be used (in addition to the standard keys) to style plot axes. Individual axes can be styled differently by using their axis name as key below the `axes` root.

```
set-style(axes: ( /* Style for all axes */ ))
set-style(axes: (bottom: ( /* Style axis "bottom" */)))
```

Axis names to be used for styling:

- School-Book and Left style:
  - `x`: X-Axis
  - `y`: Y-Axis
- Scientific style:
  - `left`: Y-Axis
  - `right`: Y2-Axis
  - `bottom`: X-Axis
  - `top`: X2-Axis

Default `scientific` Style

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.2, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
```

```
  ),
)
```

Default `school-book` Style

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.1, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
  ),
  mark: (end: ">"),
  padding: 0.4,
)
```

6.3 Chart

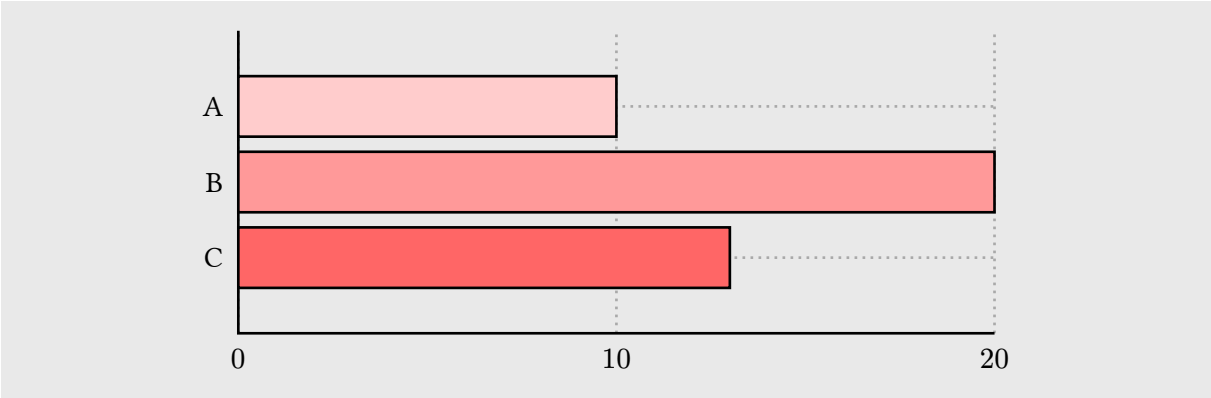
With the `chart` library it is easy to draw charts.

Supported charts are:

- `barchart(..)` and `columnchart(..)`: A chart with horizontal/vertical growing bars
  - `mode: "basic"`: (default): One bar per data row
  - `mode: "clustered"`: Multiple grouped bars per data row
  - `mode: "stacked"`: Multiple stacked bars per data row
  - `mode: "stacked100"`: Multiple stacked bars relative to the sum of a data row
- `boxwhisker(..)`: A box-plot chart

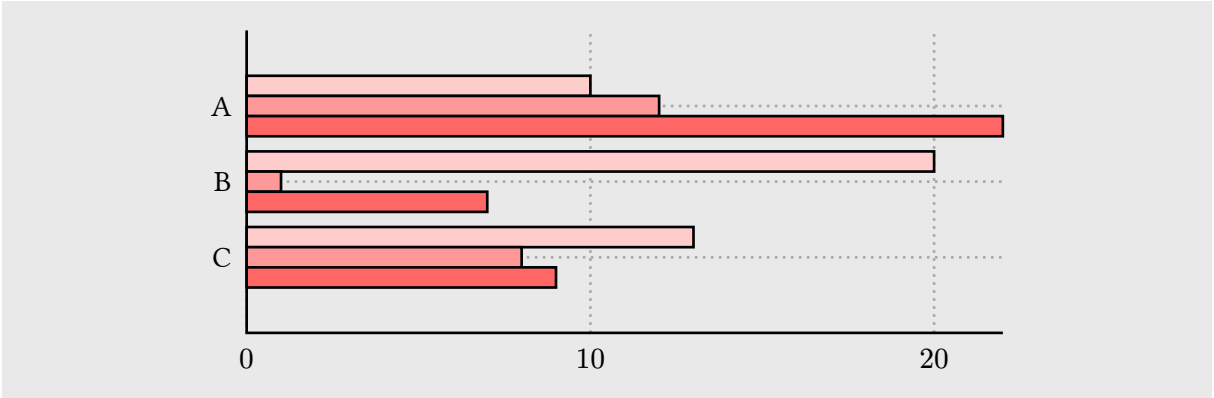
6.3.1 Examples – Bar Chart

Basic



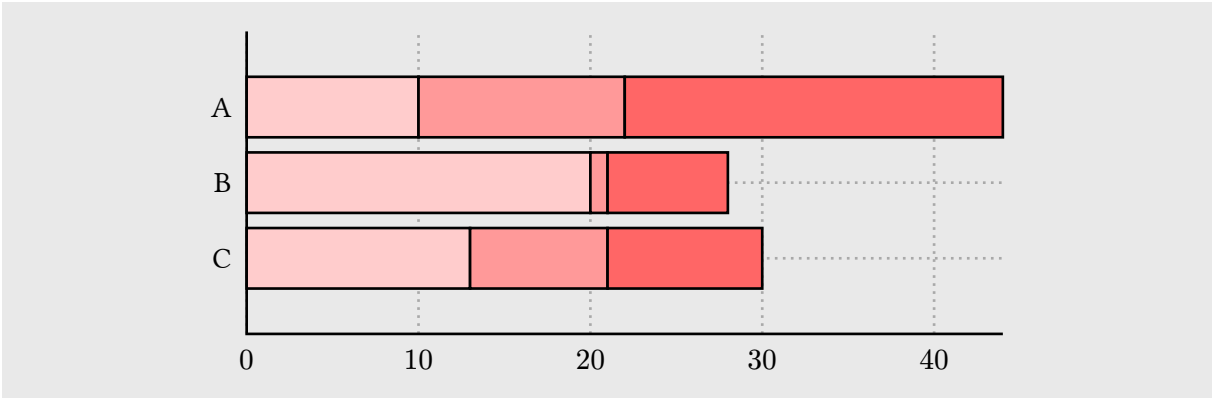

```
import cetz.chart
let data = (("A", 10), ("B", 20), ("C", 13))
chart.barchart(size: (10, auto), x-tick-step: 10, data)
```

Clustered




```
import cetz.chart
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
chart.barchart(size: (10, auto), mode: "clustered",
               x-tick-step: 10, value-key: (..range(1, 4)), data)
```

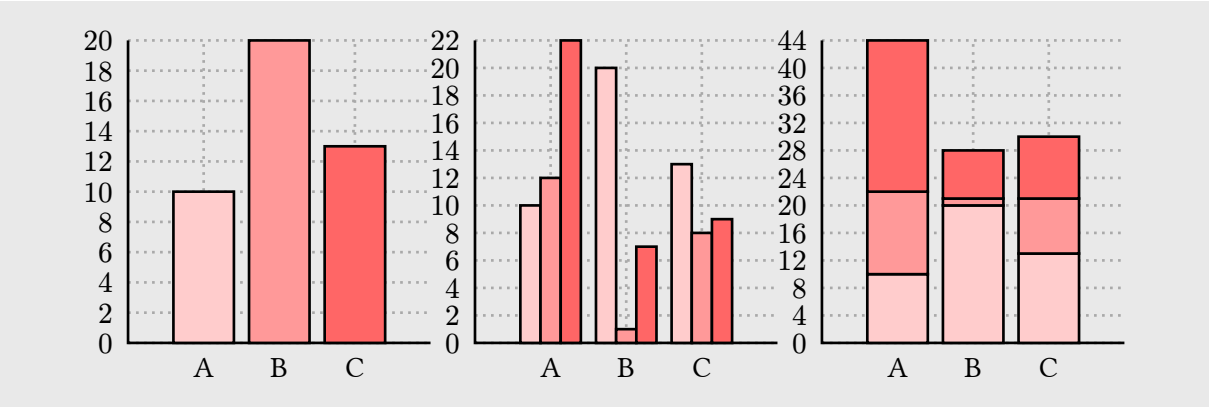
Stacked




```
import cetz.chart
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
chart.barchart(size: (10, auto), mode: "stacked",
               x-tick-step: 10, value-key: (..range(1, 4)), data)
```

6.3.2 Examples – Column Chart

Basic, Clustered and Stacked




```
import cetz.chart
// Left
let data = (("A", 10), ("B", 20), ("C", 13))
group(name: "a", {
  anchor("default", (0,0))
  chart.columnchart(size: (auto, 4), data)
})
// Center
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("a.south-east")
group(name: "b", anchor: "south-west", {
  anchor("center", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "clustered", value-key: (1,2,3), data)
})
// Right
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("b.south-east")
group(name: "c", anchor: "south-west", {
  anchor("center", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "stacked", value-key: (1,2,3), data)
})
```

- boxwhisker()

6.3.3 boxwhisker

Add one or more box or whisker plots.

Parameters

boxwhisker(
  data: `array` `dictionary`,
  size,
  y-min,
  y-max,
  label-key: `integer` `string`,
  box-width: `float`,
  whisker-width: `float`,
  mark: `string`,
  mark-size: `float`,
  ..arguments: `any`
)

data `array` or `dictionary`

Dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

See `plot.add-boxwhisker` for more details.

Examples:

- ```
  (x: 1                    // Location on x-axis
   outliers: (7, 65, 69), // Optional outliers
   min: 15, max: 60       // Minimum and maximum
   q1: 25,                // Quartiles: Lower
   q2: 35,                //            Median
   q3: 50)                //            Upper
  ```
- size (array) : Size of chart. If the second entry is auto, it automatically scales to accommodate the number of entries plotted
- y-min (float) : Lower end of y-axis range. If auto, defaults to lowest outlier or lowest min.
- y-max (float) : Upper end of y-axis range. If auto, defaults to greatest outlier or greatest max.

size

Default: `(1, auto)`

y-min

Default: `auto`

y-max

Default: `auto`

label-key `integer` or `string`

Index in the array where labels of each entry is stored

Default: `0`

box-width `float`

Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75

Default: `0.75`

whisker-width `float`

Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5

Default: `0.5`
  ```

mark `string`

Mark to use for plotting outliers. Set `none` to disable. Defaults to “x”

Default: `"*"`

mark-size `float`

Size of marks for plotting outliers. Defaults to 0.15

Default: `0.15`

..arguments `any`

Additional arguments are passed to `plot.plot`

6.3.4 Styling

Charts share their axis system with plots and therefore can be styled the same way, see Section 6.2.11.

Default `barchart` Style

```
(axes: (tick: (length: 0)))
```

Default `columnchart` Style

```
(axes: (tick: (length: 0)))
```

Default `boxwhisker` Style

```
(axes: (tick: (length: -0.1)), grid: none)
```

6.4 Palette

A palette is a function that returns a style for an index. The palette library provides some predefined palettes.

6.4.1 new

Define a new palette

A palette is a function in the form `index -> style` that takes an index (int) and returns a canvas style dictionary. If passed the string `"len"` it must return the length of its styles.

Parameters

```
new(
  stroke: stroke,
  fills: array
) -> function
```

stroke `stroke`

Single stroke style.

fills `array`

List of fill styles.

6.4.2 List of predefined palettes

- `gray`

- `red`

- `blue`

- `rainbow`

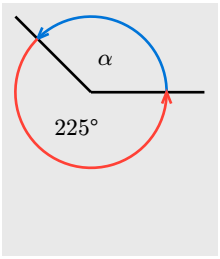
- `tango-light`

- `tango`

- `tango-dark`

6.5 Angle

The `angle` function of the angle module allows drawing angles with an optional label.




```
import cetz.angle: angle
let (a, b, c) = ((0,0), (-1,1), (1.5,0))
line(a, b)
line(a, c)
set-style(angle: (radius: 1, label-radius: .5), stroke: blue)
angle(a, c, b, label: $alpha$, mark: (end: ">"), stroke: blue)
set-style(stroke: red)
angle(a, b, c, label: n => $#{n/1deg} degree$,
  mark: (end: ">"), stroke: red, inner: false)
```

Default `angle` Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 0.25,
  mark: (
    size: 0.15,
    angle: 45deg,
    start: none,
    end: none,
    stroke: auto,
    fill: none,
```

```
  ),
)
```

6.6 Decorations

Various pre-made shapes and lines.

6.6.1 brace

Draw a curly brace between two points.

Style root: `brace`.

Anchors:

start Where the brace starts, same as the `start` parameter.

end Where the brace end, same as the `end` parameter.

spike Point of the spike, halfway between `start` and `end` and shifted by `amplitude` towards the pointing direction.

content Point to place content/text at, in front of the spike.

center Center of the enclosing rectangle.

(a-k) Debug points `a` through `k`.

Parameters

```
brace(
  start: coordinate,
  end: coordinate,
  flip: bool,
  debug: bool,
  name: string none,
  ..style: style
)
```

start `coordinate`

Start point

end `coordinate`

End point

flip `bool`

Flip the brace around

Default: `false`

debug `bool`

Show debug lines and points

Default: `false`

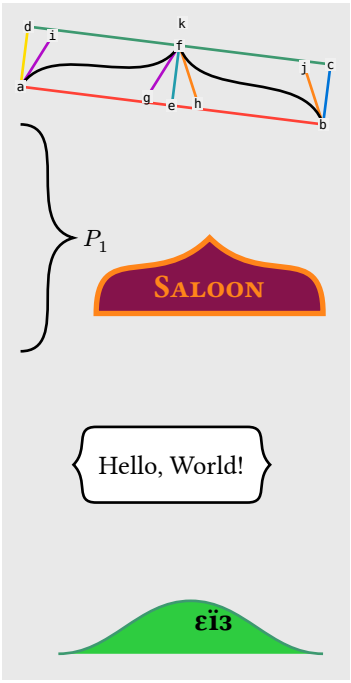
name `string` or `none`

Element name

Default: `none`

..style `style`

Style attributes




```
import cetz.decorations: brace
let text = text.with(size: 12pt, font: "Linux Libertine")

brace((0, 0), (4, -.5), pointiness: 25deg, outer-pointiness: auto,
amplitude: .8, debug: true)
brace((0, -.5), (0, -3.5), name: "brace")
content("brace.content", [$P_1$])

// styling can be passed to the underlying `merge-path` call
brace((1, -3), (4, -3), amplitude: 1, pointiness: .5, stroke: orange + 2pt,
fill: maroon, close: true, name: "saloon")
content((rel: (0, -.15), to: "saloon.center"), text(fill: orange,
smallcaps[*Saloon*]))

// as part of another path
set-origin((2, -5))
merge-path({
  brace((+1, .5), (+1, -.5), amplitude: .3, pointiness: .5)
  brace((-1, -.5), (-1, .5), amplitude: .3, pointiness: .5)
}, fill: white, close: true)
content((0, 0), text(size: 10pt)[Hello, World!])

brace((-1.5, -2.5), (2, -2.5), pointiness: 1, outer-pointiness: 1, stroke:
olive, fill: green, name: "hill")
content((rel: (.3, .1), to: "hill.center"), text[*εϊз*])
```

Styling

amplitude `<number>` (default: 0.7)

Determines how much the brace rises above the base line.

pointiness `<number> or <angle>` (default: `15deg`)

How pointy the spike should be. `0deg` or `0` for maximum pointiness, `90deg` or `1` for minimum.

outer-pointiness `<number> or <angle> or <auto>` (default: 0)

How pointy the outer edges should be. `0deg` or `0` for maximum pointiness (allowing for a smooth transition to a straight line), `90deg` or `1` for minimum. Setting this to `auto` will use the value set for `pointiness`.

content-offset `<number>` (default: 0.3)

Offset of the `content` anchor from the spike.

debug-text-size `<length>` (default: `6pt`)

Font size of displayed debug points when `debug` is `true`.

Default `brace` Style

```
(
  amplitude: 0.7,
  pointiness: 15deg,
  outer-pointiness: 0,
```

```
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

6.6.2 flat-brace

Draw a flat curly brace between two points.

This mimics the braces from TikZ's `decorations.pathreplacing` library[1]. In contrast to `brace()`, these braces use straight line segments, resulting in better looks for long braces with a small amplitude.

Style root: `flat-brace`.

Anchors:

start Where the brace starts, same as the `start` parameter.
end Where the brace end, same as the `end` parameter.
spike Point of the spike's top.
content Point to place content/text at, in front of the spike.
center Center of the enclosing rectangle.
(a-h) Debug points `a` through `h`.

Parameters

```
flat-brace(
  start: coordinate,
  end: coordinate,
  flip: bool,
  debug: bool,
  name: string none,
  ..style: style
)
```

start `coordinate`

Start point

end `coordinate`

End point

flip `bool`

Flip the brace around

Default: `false`

debug `bool`

Show debug lines and points

Default: `false`

[1] https://github.com/pgf-tikz/pgf/blob/6e5fd71581ab04351a89553a259b57988bc28140/tex/generic/pgf/libraries/decorations/pgflibrarydecorations.pathreplacing.code.tex#L136-L185

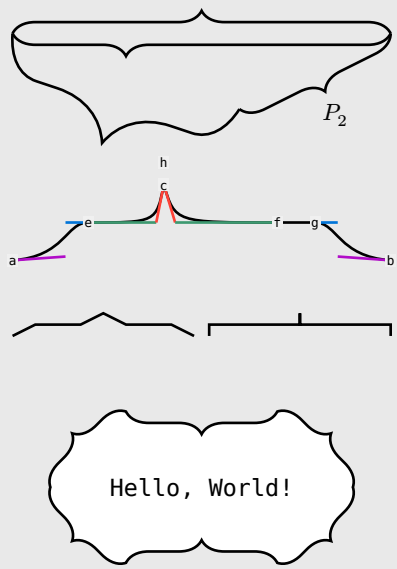
name `string` or `none`

Element name

Default: `none`

..style `style`

Style attributes




```
import cetz.decorations: flat-brace

flat-brace((), (x: 5))
flat-brace((0, 0), (5, 0), flip: true, aspect: .3)
flat-brace((), (rel: (-2, -1)), name: "a")
flat-brace((), (0, 0), amplitude: 1, curves: 1.5, outer-curves: .5)
content("a.content", [$P_2$])

flat-brace((0, -3), (5, -3), debug: true, amplitude: 1, aspect: .4,
curves: (1.5, .9, 1, .1), outer-curves: (1, .3, .1, .7))

// triangle and square braces
flat-brace((0, -4), (2.4, -4), curves: (auto, 0, 0, 0))
flat-brace((2.6, -4), (5, -4), curves: 0)

merge-path(close: true, fill: white, {
  move-to((.5, -6))
  flat-brace((), (rel: (1, 1)))
  flat-brace((), (rel: (2, 0)), flip: true, name: "top")
  flat-brace((), (rel: (1, -1)))
  flat-brace((), (rel: (-1, -1)))
  flat-brace((), (rel: (-2, 0)), flip: true, name: "bottom")
  flat-brace((), (rel: (-1, 1)))
})
content(("top.spike", .5, "bottom.spike"), [Hello, World!])
```

Styling

amplitude `<number>` (default: 0.3)

Determines how much the brace rises above the base line.

aspect `<number>` (default: 0.5)

Determines the fraction of the total length where the spike will be placed.

curves `<array> or <number>` (default: `(1, 0.5, 0.6, 0.15)`)

Customizes the control points of the curved parts. Setting a single number is the same as setting `(num, auto, auto, auto)`. Setting any item to `auto` will use its default value. The first item specifies the curve widths as a fraction of the amplitude. The second item specifies the length of the green and blue debug lines as a fraction of the curve's width. The third item specifies the vertical offset of the red and purple debug lines as a fraction of the curve's height. The fourth item specifies the horizontal offset of the red and purple debug lines as a fraction of the curve's width.

outer-curves `<array> or <number> or <auto>` (default: `auto`)

Customizes the control points of just the outer two curves (just the blue and purple debug lines). Overrides settings from `curves`. Setting the entire value or individual items to `auto` uses the values from `curves` as fallbacks.

content-offset `<number>` (default: 0.3)

Offset of the `content` anchor from the spike.

debug-text-size `<length>` (default: `6pt`)

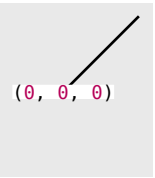
Font size of displayed debug points when `debug` is `true`.

Default `flat-brace` Style

```
(
  amplitude: 0.3,
  aspect: 0.5,
  curves: (1, 0.5, 0.6, 0.15),
  outer-curves: auto,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

7 Advanced Functions

7.1 Coordinate




```
line((0,0), (1,1), name: "l")
get-ctx(ctx => {
  // Get the vector of coordinate "l.start"
  content("l.start", [#cetz.coordinate.resolve(ctx, "l.start").at(1)], frame: "rect",
          stroke: none, fill: white)
})
```

7.2 Styles

7.2.1 resolve

Resolve the current style root

Parameters

resolve(
 current: `style`,
 new: `style`,
 root: `none` `str`,
 base: `none` `style`
)

current `style`

Current context style (`ctx.style`).

new `style`

Style values overwriting the current style (or an empty dict). I.e. inline styles passed with an element: `line(.., stroke: red)`.

root `none` or `str`

Style root element name.

Default: `none`

base `none` or `style`

Base style. For use with custom elements, see `lib/angle.typ` as an example.

Default: `none`

```
(
  fill: none,
  stroke: 1pt + luma(0%),
  radius: 1,
  mark: (
    size: 0.15,
    angle: 45deg,
    start: none,
    end: none,
    stroke: 1pt + luma(0%),
    fill: none,
  ),
)
```

```
get-ctx(ctx => {
  // Get the current line style
  content((0,0), [#cetz.styles.resolve(ctx.style, (:), root: "line")],
          frame: "rect",
          stroke: none, fill: white)
})
```