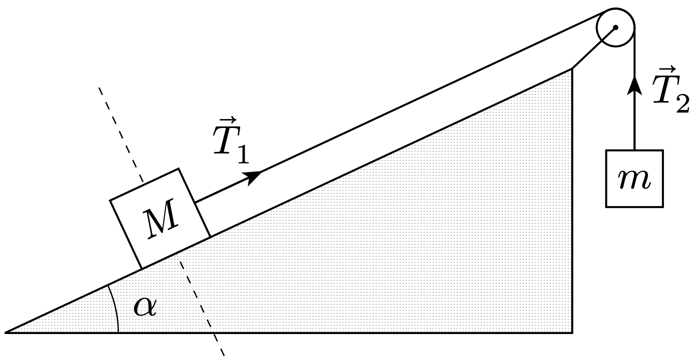
# patatrac

|pataˈtrak|



*"the funny sound of something messy suddenly collapsing onto itself"*

**Description**: This Typst package provides help with the typesetting of physics diagrams depicting classical mechanical systems. The goal:

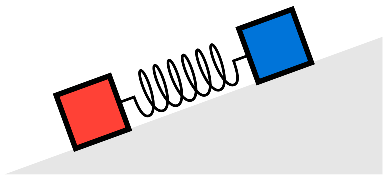
*drawing beautiful physics diagrams without trigonometry.*

The workflow is based on a strict separation between the composition and the rendering of the diagrams. The package is 100% `cetz`-compatible.

# Index

# 1 Tutorial

In this tutorial we will assume that `cetz` is the rendering engine of choice, which at the moment is the only one supported out of the box. The goal is to draw the figure below: two boxes connected by a spring laying on a sloped surface.



## 1.1 Getting started

Let's start with the boilerplate required to import `patatrac` and set up a canvas. Under the namespace `patatrac.cetz`, the package exposes a complete `cetz` version plus all cetz-based renderers.

```
1 #import "@preview/patatrac:0.0.0"
2 
3 #patatrac.cetz.canvas(length: 0.5mm, {
4   import patatrac: *
5   let draw = cetz.standard()
6 
7   // Composition & Rendering
8 })
```

At line 3, we create a new cetz canvas. At line 5, we define draw to be the cetz standard renderer provided by `patatrac` without giving any default styling option: we will go back to defaults later in the tutorial. The function `draw` will take care of outputting `cetz` elements that the canvas can print. From now on, we will only show what goes in the place of line 7, but remember that the boilerplate is still there. Let's start by adding the floor to our scene.

```
1 let floor = rect(100, 20)
2 draw(floor)
```

Line 1 creates a new patatrac `object` of type `"rect"`, which under the hood is a special function that represents our $100 \times 20$ rectangle. The output of `draw` is a cetz-element that is then picked up by the canvas and printed.

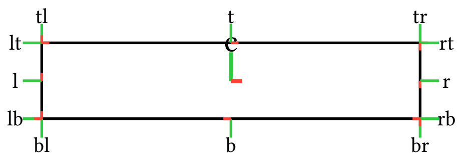
## 1.2 Introducing anchors

Every object carries with it a set of anchors. Every anchor is a point in space with a specified orientation. As anticipated, objects are functions. In particular, if you call an object on the string `"anchors"`, a complete dictionary of all its anchors is returned. For example, `floor("anchors")` gives

```
(
  c: (x: 0, y: 0, rot: 0deg),
  tl: (x: -50.0, y: 10.0, rot: 0deg),
  t: (x: 0, y: 10.0, rot: 0deg),
  tr: (x: 50.0, y: 10.0, rot: 0deg),
  lt: (x: -50.0, y: 10.0, rot: 90deg),
  l: (x: -50.0, y: 0, rot: 90deg),
  lb: (x: -50.0, y: -10.0, rot: 90deg),
  bl: (x: -50.0, y: -10.0, rot: 180deg),
  b: (x: 0, y: -10.0, rot: 180deg),
  br: (x: 50.0, y: -10.0, rot: 180deg),
  rt: (x: 50.0, y: 10.0, rot: 270deg),
  r: (x: 50.0, y: 0, rot: 270deg),
  rb: (x: 50.0, y: -10.0, rot: 270deg),
)
```

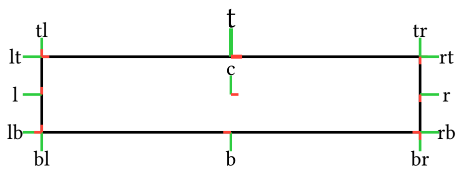
As a general rule of thumb, anchors are placed both at the vertices and at the centers of the sides of the objects and their rotations specify the tangent direction at every point. If you use the renderer `patatrac.cetz.debug` you will see exactly where and how the anchors are placed: red corresponds to the tangent (local-$x$) direction and green to the normal (local-$y$) direction.

```
let debug = cetz.debug()
draw(floor)
debug(floor)
```



As you can see, the central anchor is drawn a bit bigger and thicker. The reason is that `c` is, by default, what we call the *active anchor*. We can change the active anchor of an object by calling the object itself on the name of the anchor. For example if we instead draw the anchors of the object `floor("t")` what we get is the following.

```
let debug = cetz.debug()
draw(floor)
debug(floor("t"))
```



When doing so, we have to remember that Typst functions are pure: don't forget to reassign your objects if you want the active anchor to change "permanently"!

## 1.3 Composition

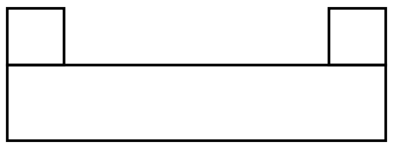
Now, let's add in the two blocks. First of all, we need to place the blocks on top of the floor. To do so we use the `place` function which takes two objects and gives as a result the first object translated such that its active anchor location overlaps with that of the second object.

```
let floor = rect(100, 20)

let A = rect(15, 15)
let B = rect(15, 15)

A = place(A("bl"), floor("tl"))
B = place(B("br"), floor("tr"))

draw(floor, A, B)
```



Now we should move the blocks a bit closer and add the spring.

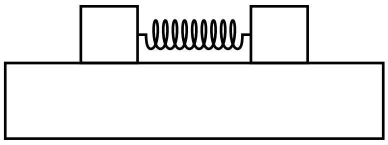
```
let floor = rect(100, 20)
let A = rect(15, 15)
let B = rect(15, 15)

A = place(A("bl"), floor("tl"))
B = place(B("br"), floor("tr"))

A = move(A, +20, 0)
B = move(B, -20, 0)

let k = spring(A("r"), B("l"))

draw(floor, A, B, k)
```

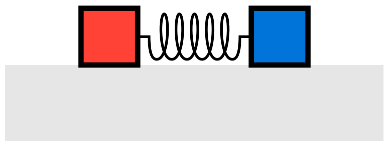


## 1.4 Styling

The styling is pretty self-explanatory. The only thing to notice is that objects drawn with the same call to `draw` share the same styling options, therefore multiple calls to `draw` are required for total stylistic freedom.

```
// ...

draw(floor, fill: luma(90%), stroke: none)
draw(k, radius: 6, pitch: 4, pad: 3)
draw(A, stroke: 2pt, fill: red)
draw(B, stroke: 2pt, fill: blue)
```



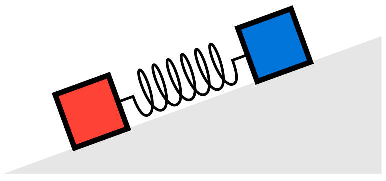
Remember that since you are inside a `cetz` canvas you are free to add whatever detail you like to make your picture more expressive. This picture is nice but drawing it without `patatrac` wouldn't have been much harder. I want you to see where `patatrac` shines so `stick` with me while I exchange the floor for an incline.

```
let floor = incline(100, 20deg)
let A = rect(15, 15)
let B = rect(15, 15)

A = stick(A("bl"), floor("tl"))
B = stick(B("br"), floor("tr"))

A = slide(A("c"), +20, 0)
B = slide(B("c"), -20, 0)

// ...
```



What have I done? At line 1, I used an `incline` instead of a rectangle which I create by giving its width and steepness. Then, at lines 5 and 6, I replaced the calls to `place` with an identical call to `stick`. This function, instead of simply translating the object, also rotates it to make sure that its active anchor faces the second anchor. By doing so, I'm sure that the two blocks rest on the incline correctly. Then, at lines 8 and 9, I replaced the calls to `move` with identical (up to a change of active anchors) calls to `slide`. This function, instead of translating the objects in the global coordinate system, translates them inside the rotated coordinate system of their active anchors. By doing so, I make the two blocks slide along the incline surface.

## 1.5 Defaults

The picture is done but there's another thing I'd like to show you and that will come in handy for more complex pictures. As promised, we have to go back to the boilerplate. Do you remember the line where we defined `draw`? Instead of specifying `stroke: 2pt` every time we draw a rectangle, we can put inside the call to `patatrac.cetz.standard` the information that by default all rectangles should have `2pt` of stroke. Even if we have only one spring it makes sense to do the same for the styling options of `k`, so that if we create a second spring it will look the same.

```
let draw = cetz.standard(
  rect: (stroke: 2pt),
  spring: (radius: 6, pitch: 4, pad: 3),
)

// ...

draw(floor, fill: luma(90%), stroke: none)
draw(k)
draw(A, fill: red)
draw(B, fill: blue)
```

Here is the full code.

```
#import "@preview/patatrac:0.0.0"

#patatrac.cetz.canvas(length: 0.5mm, {
  import patatrac: *
  let draw = cetz.standard(
    rect: (stroke: 2pt),
    spring: (radius: 6, pitch: 4, pad: 3),
  )

  let floor = incline(100, 20deg)
  let A = rect(15, 15)
  let B = rect(15, 15)

  A = stick(A("bl"), floor("tl"))
  B = stick(B("br"), floor("tr"))

  A = slide(A("c"), +20, 0)
  B = slide(B("c"), -20, 0)

  let k = spring(A("r"), B("l"))

  draw(floor, fill: luma(90%), stroke: none)
  draw(k)
  draw(A, fill: red)
  draw(B, fill: blue)
})
```

Okay, now that we have the final drawing we can spend a few words to clarify what’s going on. Read Section 2 to understand better.

# 2 Core system

The whole `patatrac` package is structured around three things:

1. anchors,
2. objects,
3. renderers.

Let's define them one by one.

## 2.1 Anchors

Anchors are simply dictionaries with three entries `x`, `y` and `rot` that are meant to specify a 2D coordinate system. The values associated with `x` and `y` are either lengths or numbers and the package assumes that this choice is unique for all the anchors used in the drawing. These two entries specify the origin of the local coordinate system on the canvas. `rot` on the other hand always takes values of type `angle` and specifies the direction in which the local-x axis is pointing. Whenever `patatrac` expects the argument of a method to be an anchor it automatically calls `anchors.to-anchor` on that argument. This allows you, the end user, to specify anchors in many different styles:

- `(x: ..., y: ..., rot: ...)`,
- `(x: ..., y: ...)`,
- `(..., ..., ...)`,
- `(..., ...)`.

All options where the rotation is not specified default to `0deg`. Moreover, objects can automatically be converted to anchors: `to-anchor` simply results in the object's active anchor. The local coordinate system is right-handed if the positive $z$-direction is taken to point from the screen towards our eyes.

## 2.2 Objects

Objects are special functions created with a call to an object constructor. All object constructors ultimately reduce to a call to `object`, so that all objects behave in the same way. The result is a callable function, let's call it `obj`, such that:

- `obj()` returns the active anchor,
- `obj("anchor-name")` returns an equivalent object but with the specified anchor as active,
- `obj("anchors")` returns the full dictionary of anchors,
- `obj("active")` returns the key of the active anchor,

- `obj("type")` returns the object type,
- `obj("data")` returns the carried metadata,
- `obj("repr")` returns a dictionary representation of the object meant only for debugging purposes.

If you want to create an object from scratch all you need to do is to use the `object` constructor yourself.

```
let custom = patatrac.objects.object(
  "custom-type-name",
  "active-anchor-name",
  dictionary-of-anchors,
  data: payload-of-metadata
)
```

### 2.3 Renderers

A renderer is a function whose job is to take default styling options and return a function capable of rendering objects. This function will take one or more objects, associate each object to a drawing function according to the object's type and return the rendered result, all of this while taking care of any styling option. The journey from a set of drawing functions to an actual drawing starts with a call to `renderer`.

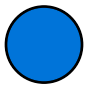
```
let my-renderer = patatrac.renderer((
  // drawing functions
  rect: (obj, style) => { ... },
  circle: (obj, style) => { ... },
  ...
))
```

For example, this is the way in which `patatrac.cetz.standard` is defined. `my-renderer` is not yet ready to render stuff: we need to specify any default styling option. We do this by calling `my-renderer` itself.

```
let draw = my-renderer(rect: (stroke: 2pt))
```

The variable `draw` is the function we use to actually render objects. This step where we provide defaults is kept separate from the call to `renderer` so that the end user can put his own defaults into the renderer: the developer should expose `my-renderer` and not `draw`. Defaults that are set by the developer can simply be hardcoded inside the drawing functions definitions; and this is exactly how the package does for its own renderers. Now, use `draw` to print things.

```
draw(circle(10), fill: blue)
```



If you want, you can extract from `my-renderer` the full dictionary of drawing functions that was used to for its definition.

```
my-renderer("functions")
```

```
(rect: (..) => .., circle: (..) => ..)
```

This allows the user to extend, modify and combine existing renderers if needed. For example, we could start from the `cetz.standard` renderer and override the algorithm for drawing circles.

```
let my-renderer = renderer(cetz.standard("functions") + (
  circle: (obj, style) => { ... }
))
```

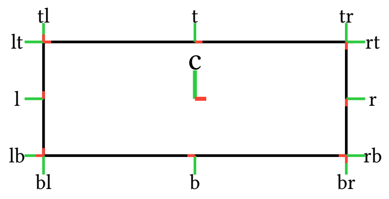
# 3 Objects one by one

We will now go through the various object constructors one by one. In what follows, we will omit the boilerplate and the rendering stage, in order to focus on composition.

## 3.1 Rectangles

The constructor `rect(width, height)` takes only the width and the height of the rectangle.

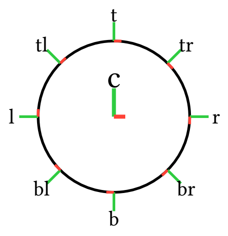
```
rect(80,300)
```



## 3.2 Circles

The constructor `circle(radius)` takes only the radius of the circle.

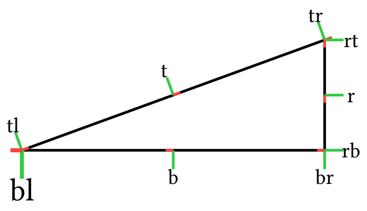
```
circle(20)
```



## 3.3 Inclines

The constructor `incline(width, angle)` takes the incline width and the angle between base and hypotenuse.

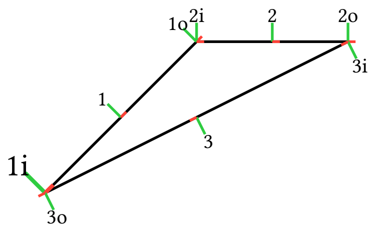
```
incline(80, 20deg)
```



## 3.4 Polygons

The constructor `polygon(..args)` takes a series of anchors representing in clockwise order the vertices of a 2D shape.

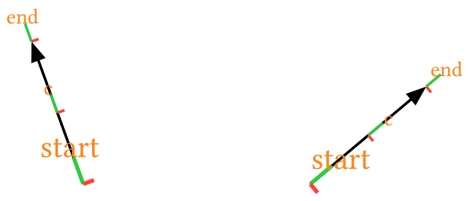
```
polygon((0,0), (40,40), (80, 40))
```



## 3.5 Arrows

The constructor `arrow(start, length, angle: none)` takes an anchor and a length. The arrow is located at the anchor's location and oriented towards the positive $y$ direction of the specified anchor. If the named parameter `angle` is set to something different from `none`, the arrow's orientation is instead determined by its value.

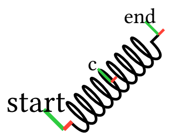
```
arrow((0,0,20deg), 20)
arrow((30,0,20deg), 20, angle: 40deg)
```



## 3.6 Springs

The constructor `spring(start, end)` takes two anchors whose location determines where the spring starts and ends.

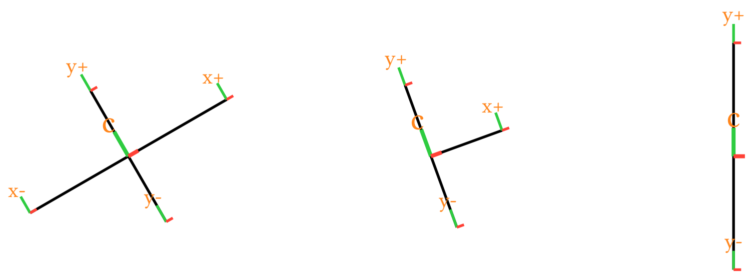
```
spring((0,0), (25,25))
```



## 3.7 Axes

The constructor `axes(at, xlength, ylength, rot: true)` takes an anchor and two lengths, for the two axes. The specified lengths are one sided, meaning that the total axis length is double that. If an axis extends into the positive and negative directions by different amounts its length must be specified as an array `(negative-extension, positive-extension)`.

```
axes((0,0,30deg), 20, 10)
axes((80,0,20deg), (0, 20), 10)
axes((160,0,10deg), 0, 30)
```

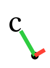


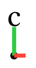
As seen in the last example, a named parameter `rot` can be set to `false` to make the constructor ignore the anchor's rotation.

## 3.8 Points

The constructor `point(at, rot: true)` takes a single anchor. A named parameter `rot` can be set to `false` to make the constructor ignore the anchor's rotation.

```
point((0,0,30deg))
point((50,0,30deg), rot: false)
```





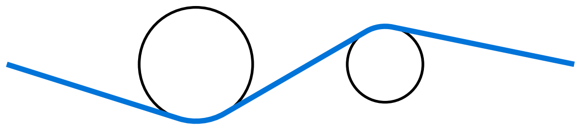
## 3.9 Ropes

The main idea behind how ropes work is the following:

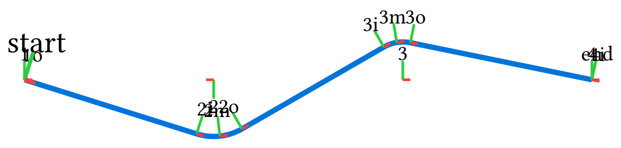
*ropes are one dimensional strings that wrap around anchors and circles.*

The constructor `rope(..args)` requires the list of anchors and circles that you want the rope to wrap around. Since there are two ways in which any given rope can wrap around a circle, the rotation of the active anchor of the circle will tell the rope from which direction to start "orbiting" around the circle.

```
let C1 = circle(15)
let C2 = place(circle(10), (50, 0))
let R = rope((-50, 0), C1("b"), C2("t"), (+100, 0))
```



Ropes provide many different anchors. Anchors are named with increasing whole numbers starting from one converted to strings and eventually followed by a letter `"i"`, `"m"`, `"o"`. The letter `"i"` specifies that we are either starting an arc of circumference around a circle or approaching a vertex. The letter "m" denotes anchors which are placed at the middle of an turn. The letter "o" is placed at the end of anchors that either specify the outgoing direction from a vertex or the last point on an arc. There are also two special anchors start and end, whose name is self-explanatory. Here is the full list of anchors for the previous example.

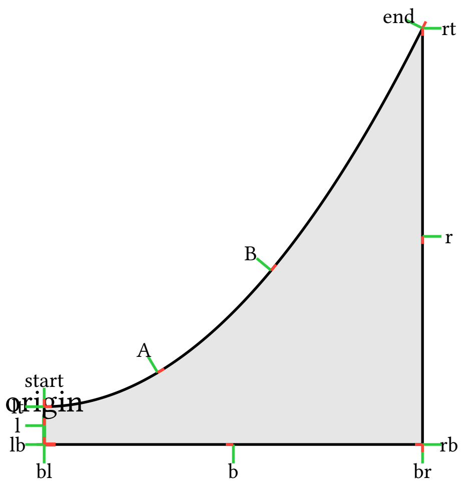


```
(
  "1": (x: -50, y: 0, rot: 0deg),
  "1o": (x: -50, y: 0, rot: -17.46deg),
  "2": (x: 0, y: 0, rot: 180deg),
  "2i": (x: -4.5, y: -14.309088021254185, rot: 342.54deg),
  "2m": (
    x: 1.638519763245789,
    y: -14.910239870151418,
    rot: 6.27deg,
  ),
  "2o": (
    x: 7.500000000000002,
    y: -12.990381056766578,
    rot: 30deg,
  ),
  "3": (x: 50.0, y: 0.0, rot: 0deg),
  "3i": (x: 45.0, y: 8.660254037844387, rot: 30deg),
  "3m": (
    x: 48.39575777115576,
    y: 9.87048159266775,
    rot: 9.23deg,
  ),
  "3o": (x: 52.0, y: 9.797958971132712, rot: 348.46deg),
  "4": (x: 100, y: 0, rot: 0deg),
  "4i": (x: 100, y: 0, rot: 348.46deg),
  start: (x: -50, y: 0, rot: -17.46deg),
  end: (x: 100, y: 0, rot: 348.46deg),
)
```

## 3.10 Terrains

Terrains are objects that you can use to create arbitrarily shaped surfaces. The constructor `terrain(fn, range, scale: 1, epsilon: 0.001%, ..stops)` takes a function describing the profile and its domain, expressed as a tuple `(min, max)`. The named parameter `scale` is a prefactor that is applied to the coordinates before calculating the anchors' positions and before drawing.

```
let ground = terrain(
  x => 0.1 + x*x, (0, 1),
  scale: 100, A: 30%, B: 0.6,
)
```

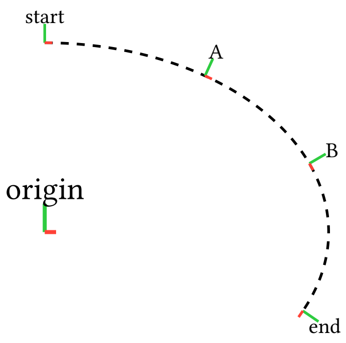


Here we are also specifying two points $A$ and $B$, by giving their position on the specified range either as number or ratio. These points are automatically added as anchors tangent to the surface. In order to rotate the anchors correctly `patatrac` needs to differentiate numerically the given function. The named parameter `epsilon` specifies the step size for the incremental ratio used to approximate the derivative.

### 3.11 Trajectories

Trajectories are objects that you can use to create arbitrarily shaped curves. The constructor `trajectory(fn, range, scale: 1, epsilon: 0.001%, ..stops)` takes a function parametrizing the curve and its domain, expressed as a tuple `(min, max)`.

```
let motion = trajectory(
  t => (1.5*calc.sin(t), calc.cos(t)), (0, 2),
  scale: 50, A: 30%, B: 1.2,
)
```



Here we are also specifying two points $A$ and $B$, by giving their position on the specified range either as number or ratio. These points are automatically added as anchors tangent to the curve. In order to rotate the anchors correctly `patatrac` needs to differentiate numerically the given function. The named parameter `epsilon` specifies the step size for the incremental ratio used to approximate the derivative.

# 4 Renderers

## 4.1 `patatrac.cetz.debug`

This renderer is capable of rendering objects of the following types: `arrow`, `axes`, `circle`, `incline`, `point`, `polygon`, `rect`, `rope`, `spring`, `terrain`, `trajectory`. The renderer should be used for debugging purposes only. Independently of the object type, the styling options are

- `length`: length of the tangent line,
- `stroke`: stroke used to draw the tangent and normal lines,
- `fill`: fill color for the label with the anchor's name.

## 4.2 `patatrac.cetz.standard`

This renderer is capable of rendering objects of the following types: `arrow`, `axes`, `circle`, `incline`, `point`, `polygon`, `rect`, `rope`, `spring`, `terrain`, `trajectory`. At the moment, this is the only renderer capable of producing final drawings. We will now list all styling options available for each object type, together with a brief description. (These lists are written by hand, please report any mistake.)

`arrow`:

- `stroke`: stroke used to draw the arrow body,
- `mark`: cetz-style mark used to specify how the tail and the tip of the arrow should look.

`axes`:

- `stroke`: x and y axes stroke,
- `mark`: x and y axes cetz-style mark,
- `xstroke`: x axis stroke,
- `xmark`: x axis mark,
- `ystroke`: y axis stroke,
- `ymark`: y axis mark.

`circle`:

- `stroke`: outline stroke,
- `fill`: circle's fill,

`incline`:

- `stroke`: outline stroke,

- `fill`: incline’s fill,

`point`:

- `radius`: radius of the dot,
- `stroke`: stroke of the dot,
- `fill`: fill of the dot,
- `label`: content of the label,
- `align`: alignment of the label with respect to the center of the dot (technically it’s the opposite but the label position is what is changing),
- `lx`: after-alignment shift of the label in the global x direction,
- `ly`: after-alignment shift of the label in the global y direction.

`polygon`:

- `stroke`: outline stroke,
- `fill`: polygon’s fill.

`rect`:

- `stroke`: outline stroke (expressed in the style of `std.rect`, see here),
- `fill`: rectangle’s fill,
- `radius`: how much to round the corners (expressed in the style of `std.rect`, see here).

`rope`:

- `stroke`: rope’s stroke,

`spring`:

- `stroke`: curve’s stroke,
- `pitch`: distance between rings,
- `n`: number of rings,
- `pad`: minimum amount of space between the start and end points of the curve which must be occupied, on each side, by a linear segment,
- `radius`: radius of the rings,
- `curliness`: how much the rings are visible (either a `ratio` or `none` to indicate that a zig-zag pattern should be used instead of the default coil-like shape),

`terrain`:

- `stroke`: outline's stroke,
- `fill`: terrain's fill,
- `epsilon`: distance in domain-space between the points used to approximate the graph (can be a `ratio` or either a number or a length, matching the units used for the domain of the function),
- `smooth`: whether to use a hobby curve to smooth between points or interpolate linearly (boolean).

`trajectory`:

- `stroke`: outline's stroke,
- `epsilon`: distance in domain-space between the points used to approximate the curve (can be a `ratio` or either a number or a length, matching the units used for the domain of the function),
- `smooth`: whether to use a hobby curve to smooth between points or interpolate linearly (boolean).

# 5 Useful lists

In this section you'll find a few useful lists. These lists are generated semi-automatically.

## 5.1 All renderers

This is the complete list of available renderers

- `patatrac.cetz.debug`
- `patatrac.cetz.standard`

## 5.2 All objects and related functions

Here is the list of all object constructors. These are all available directly under the namespace `patatrac`.

- `arrow(start, length, angle: none)`
- `axes(at, xlength, ylength, rot: true)`
- `circle(radius)`
- `incline(width, angle)`
- `point(at, rot: true)`
- `polygon(..args)`
- `rect(width, height)`
- `rope(..args)`
- `spring(start, end)`
- `terrain(fn, range, scale: 1, epsilon: 0.001%, ..stops)`
- `trajectory(fn, range, scale: 1, epsilon: 0.001%, ..stops)`

Here is the list of all object related functions. These are all available directly under the namespace `patatrac`.

- `alias(constructor, new-type)`. Turns an object constructor into a new constructor that creates identical objects but with a different type. This is useful when there are two objects that share the same logic but have very different rendering routines. This helps reduce the amount of styling options passed to renderers.
- `match(obj, target, x: true, y: true, rot: true)`. Translates and rotates an object to ensure that the active anchor of `obj` becomes equal to the `target`ed anchor. The named parameters `x`, `y` and `rot` take boolean values that determine if the corresponding anchor parameter

can be changed or not: `true` is the default and `false` means that the parameter has to remain fixed.

- `move(obj, dx, dy, rot: 0deg)`. Translates the object in global coordinates. Its equivalent to `slide.with(rot: 0deg)`.
- `object(obj-type, active, anchors, data: none)`. Creates an object.
- `rotate(obj, angle, ref: none)`. Rotates the object around a given anchor by the specified angle. The anchor is taken to be the active anchor of the object if `ref` is `none` otherwise `ref` itself is used as origin.
- `slide(obj, dx, dy, rot: none)`. Translates the object in a rotated coordinate system. If `rot` is `none`, the coordinate system is rotated like the active anchor of the object. If `rot` is a specified angle, it will be used as the reference frame rotation.
- `stick(obj, target)`. Translates and rotates an object to ensure that the active anchor of the object becomes equal in origin and opposite in direction with respect to the `target`ed anchor.
- `transform(obj, func)`. Applies a given function to all the anchors of an object or all the anchors of all the objects in a group of objects. The result is the same object or group but with the operation applied. Nested groups are preserved.

## 5.3 Anchors related functions

Under the namespace `patatrac.anchors` you can find

- `anchor(x, y, rot)`. Every anchor represents a coordinate system where (x,y) is the origin of the local coordinate system and `rot` is the angle between the paper left-to-right axis and the anchor's x-axis. The local y-axis is always directed 90deg anticlockwise with respect to local x-axis. Most often, anchors are use to describe points on surfaces. In that case, the local x-axis is the tangent to the surface and the local y-axis is the outgoing normal to the surface.
- `distance(anchor1, anchor2)`. Returns the distance between two anchors.
- `lerp(anchor1, anchor2, by, rot: true)`. Linear interpolation between anchors. The field `by` is a `ratio`. If `rot` is set to `true` the result's rotation is fixed to the first anchor's rotation.
- `move(anchor1, dx, dy)`. Moves an anchor in the global coordinate system.
- `pivot(target, origin, angle, rot: true)`. The `pivot` function returns the anchor you get if you take the `target` anchor and rotate it

around the `origin` location by `angle`. If `rot` is set to `false` the anchor's rotation is fixed to the rotation of `target`.

- `rotate(target, angle)`. Rotates an anchor by some angle.
- `slide(target, dx, dy, rot: none)`. Moves an anchor in its own rotated coordinate system.
- `term-by-term-difference(anchor1, anchor2)`. Returns the term by term difference `anchor1 - anchor2`. Named arguments lock one coordinate to the first anchor's value.
- `term-by-term-sum(anchor1, anchor2)`. Returns the term by term sum of two anchors. Named arguments lock one coordinate to the first anchor's value.
- `to-anchor(thing, panic: true)`. Tries to convert anything to an anchor. By default, the function panics if `thing` cannot be converted to an anchor, but if the named parameter `panic` is set to `false` it will silently return `none`, instead of panicking.
- `x-inter-x(anchor1, anchor2)`. Computes the intersection between two anchors' tangent lines. The output orientation is the first anchor's orientation. This function is effectively translating the first anchor along its tangent line to meet the second anchor's tangent line. The result is `none` if there's no intersection, otherwise the result is an anchor. Named arguments lock one coordinate to `anchor1`'s value.
- `x-inter-y(anchor1, anchor2)`. Computes the intersection between the first anchors' tangent line and the second anchors' normal line. The output orientation is the first anchor's orientation. This function is effectively translating the first anchor along its tangent line to meet the second anchor's normal line. The result is `none` if there's no intersection, otherwise the result is an anchor. Named arguments lock one coordinate to `anchor1`'s value.
- `x-look-at(anchor1, anchor2)`. Rotates the first anchor such that its x axis points towards the second anchor.
- `x-look-from(anchor1, anchor2)`. Rotates the first anchor such that its x axis points opposite to the second anchor.
- `y-inter-x(anchor1, anchor2)`. Computes the intersection between the first anchors' normal line and the second anchors' tangent line. The output orientation is the first anchor's orientation. This function is effectively translating the first anchor along its normal line to meet the second anchor's tangent line. The result is `none` if there's no inter-

section, otherwise the result is an anchor. Named arguments lock one coordinate to `anchor1`'s value.

- `y-inter-y(anchor1, anchor2)`. Computes the intersection between two anchors' normal lines. The output orientation is the first anchor's orientation. This function is effectively translating the first anchor along its normal line to meet the second anchor's normal line. The result is `none` if there's no intersection, otherwise the result is an anchor. Named arguments lock one coordinate to `anchor1`'s value.
- `y-look-at(anchor1, anchor2)`. Rotates the first anchor such that its y axis points towards to the second anchor.
- `y-look-from(anchor1, anchor2)`. Rotates the first anchor such that its y axis points opposite to the second anchor.