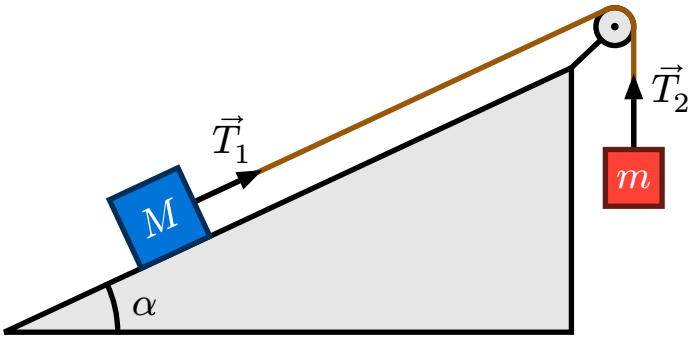
# patatrac

|pata'trak|




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

## 1 Introduction

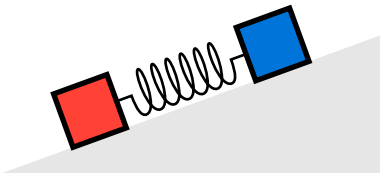
This Typst package provides help with the typesetting of physics diagrams depicting classical mechanical systems. The goal:

*drawing beautiful physics diagrams without trigonometry.*

The workflow is based on a strict separation between the composition and the rendering (drawing) of the diagrams. The composition stage is 100% agnostic of the rendering engine used for drawing. A `cetz`-based renderer is provided.

## 2 Tutorial

In this tutorial we will assume that `cetz` is the rendering engine of choice, which for 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.



Let's start with the boilerplate required to import `patatrac` and `cetz`.

```
1  #import "@preview/cetz:0.3.4": canvas
2  #import "@preview/patatrac:0.0.0"
3
4  #canvas(length: 0.5mm, {
5    import patatrac: *
6    let draw = renderers.cetz.standard()
7
8    // Composition & Rendering
9    ()
10 }.flatten())
```

At line 4, we define draw to be the cetz standard renderer provided by `patatrac` without providing any default styling option: we will do it later. 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 9, 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.

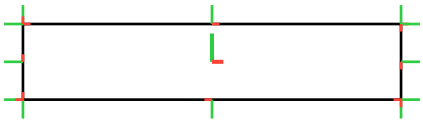
Every object carries with it a set of anchors. Every anchor is a point in space with a specified orientation. As anticipated, objects are functions. In particular, if you call an object on the string `"anchors"`, a complete dictionary of all its anchors is returned. For example, `floor("anchors")` gives

```
(
  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),
)
```

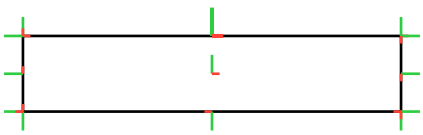
The anchors are placed both at the vertices and at the centers of the faces of the rectangle and their rotations specify the tangent direction at every point. If you pay attention you will see that the rotation of the anchors is an angle which increases as one rotates counter-clockwise and with zero corresponding to the right direction. If you use the renderer `patatrac.renderers.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 = renderers.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 = renderers.cetz.debug()
draw(floor)
debug(floor("t"))
```

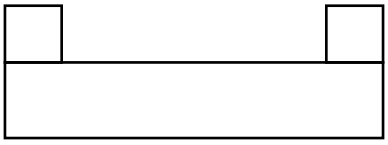


When doing so, we have to remember that Typst functions are pure: don't forget to reassign your objects if you want the active anchor to change "permanently"!

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.

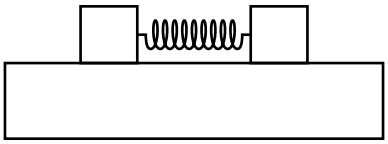
```
let floor = rect(100, 20)

```

```
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)
```



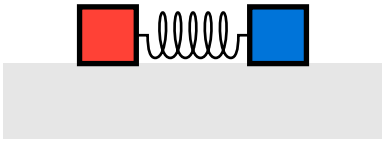
Now we should move the blocks a bit closer and add the spring.

```
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)
```



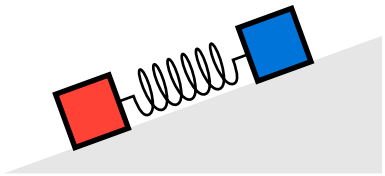
The styling is pretty self-explanatory. The only thing to notice is that objects drawn with the same call to `draw` share the same styling options, therefore multiple calls to `draw` are required for total stylistic freedom.

```
// ...

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)
```



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.

```
1  let floor = incline(100, 20deg)
2  let A = rect(15, 15)
3  let B = rect(15, 15)
4
5  A = stick(A("bl"), floor("tl"))
6  B = stick(B("br"), floor("tr"))
7
8  A = slide(A("c"), +20, 0)
9  B = slide(B("c"), -20, 0)
10
11 // the rest is the same
```



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.

The picture is done but we can improve the code a bit. As promised, we have to go back to the boilerplate. Do you remember the line where we defined `draw`? We can put inside the call to `patatrac.renderers.cetz.standard` the information that all rectangles should have `2pt` of stroke and get rid of this information from the calls to `draw` for `A` and `B`. Even if we have only one spring it makes sense to do the same for the styling options of `k`.

```
let draw = renderers.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/cetz:0.3.4": canvas
#import "@preview/patatrac:0.0.0"

#canvas(length: 0.5mm, {
  import patatrac: *
  let draw = renderers.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)

}.flatten())
```

Okay, now that we have the final drawing we can spend a few words to clarify what's going on. Read Section 3 to understand better.

# 3 System

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

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

Let's define them one by one.

***1. 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 end 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 `anchor.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. 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.

***3. Renderers*** are special functions created with a call to `renderer`. A renderer is essentially a machine that takes one or more objects, associates each object to a drawing function according to the object's type and returns the rendered result. If you want to retrieve the dictionary of type-function pairs call the renderer without providing any argument. If you specify named arguments, the renderer will pass them to the drawing functions as styling options.

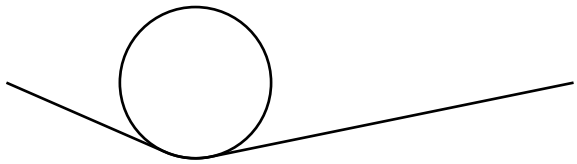
# 4 Ropes

Normally, drawing Atwood machines tends to be really cumbersome, but with `patatrack` pulleys are extremely easy to draw, thanks to the mechanics of `ropes`. The main idea behind how ropes work is the following:

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

To create a rope all you have to do is to provide 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. An example will make everything very clear.

```
let C = circle(15)
let R = rope((-50, 0), C("b"), (+100, 0))
draw(C, R)
```



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: -23.58deg),
  "2": (x: 0, y: 0, rot: 180deg),
  "2i": (
    x: -7.999999999999999,
```

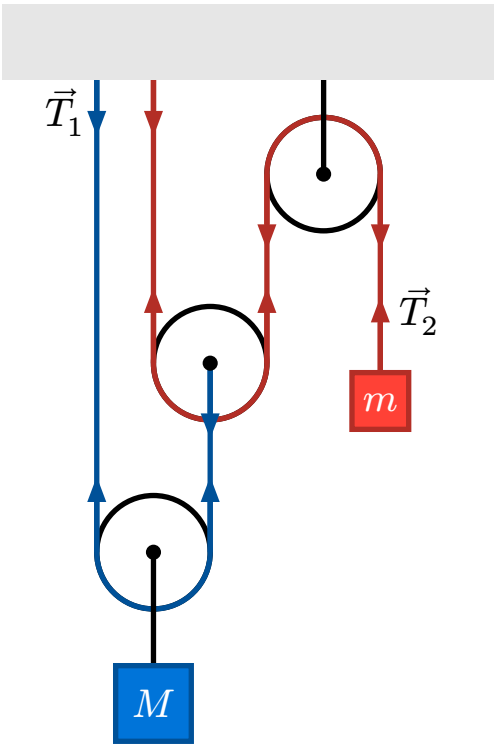
```
    y: -18.33030277982336,
    rot: 336.42deg,
  ),
  "2m": (
    x: -2.0977238700197924,
    y: -19.889684627091228,
    rot: 353.98deg,
  ),
  "2o": (
    x: 3.9999999999999982,
    y: -19.595917942265427,
    rot: 11.54deg,
  ),
  "3": (x: 100, y: 0, rot: 0deg),
  "3i": (x: 100, y: 0, rot: 11.54deg),
  start: (x: -50, y: 0, rot: -23.58deg),
  end: (x: 100, y: 0, rot: 11.54deg),
)
```

Really, there isn't anything more to say about ropes: they just work. Check out the following example to see te full potential of ropes into action. Notice how little code is required for the diagram composition: less that 30 lines of code.

```
import "@preview/patatrac:0.0.0" as patatrac: *
let draw = patatrac.renderers.cetz.standard()

// Composition

let ceiling = move(rect(130, 20), 30, 5)
let radius = 15

let C1 = move(circle(radius), 50, -30)
let A = move(place(rect(15, 15), C1("r")), 0, -60)
let L1 = rope(C1, anchors.y-inter-x(C1, ceiling("bl")))

let C2 = circle(radius)
C2 = stick(C2("r"), C1("l"))
C2 = move(C2, 0, -50)

let C3 = circle(radius)
C3 = place(C3("r"), C2("c"))
C3 = move(C3, 0, -50)

let rope23 = rope(C2("c"), C3, (C3("l")().x, 0))
let rope12 = rope(
  A("c"), C1("r"), C2("r"), (C2("l")().x, 0)
)

```

```
let B = rect(20, 20)
B = place(B, C3("c"))
B = move(B, 0, -40)
let ropeB = rope(B, C3("c"))

// Rendering

draw(C1, C2, C3, stroke: 2pt)
draw(rope12, stroke: 2pt + red.darken(30%))
draw(rope23, stroke: 2pt + blue.darken(30%))
draw(L1, ropeB, stroke: 2pt)
draw(A, fill: red, stroke: 2pt + red.darken(30%))
draw(B, fill: blue, stroke: 2pt + blue.darken(30%))

let tensionA1 = arrow(A("t"), 20)
let tensionA2 = arrow(C1("r"), 20, angle: -90deg)
draw(stroke: 2pt + red.darken(30%),
  tensionA1, tensionA2,
  place(tensionA1, C2("r")),
  place(tensionA1, C2("l")),
  place(tensionA2, rope12("end")),
  place(tensionA2, C1("l")),
)

let tensionB1 = arrow(rope23("start"), 20, angle: -90deg)
let tensionB2 = arrow(C3("r"), 20, angle: +90deg)
draw(stroke: 2pt + blue.darken(30%),
  tensionB1,
  place(tensionB1, rope23("end")),
  tensionB2,
  place(tensionB2, C3("l"))
)

draw(
  label: math.arrow($T_1$),
  align: top + right,
  lx: -3, ly: -10,
  point(rope23("end"), rot: false)
)
draw(
  label: math.arrow($T_2$),
  align: left,
  lx: 5, ly: -5,
  point(tensionA1("end"), rot: false)
)
draw(radius: 2,
  point(C1("c")),
  point(C2("c")),
```

```
  point(C3("c"))
)
draw(point(A("c")), label: text(fill: white, $m$), ly: 1)
draw(point(B("c")), label: text(fill: white, $M$))
draw(ceiling, fill: luma(90%), stroke: none)
```




Assuming the system is at equilibrium, in the previous picture, arrows lengths are not to scale with the actual tensions magnitudes!

# 5 Lists

In this section you'll find a few useful lists (made by hand, expect errors).

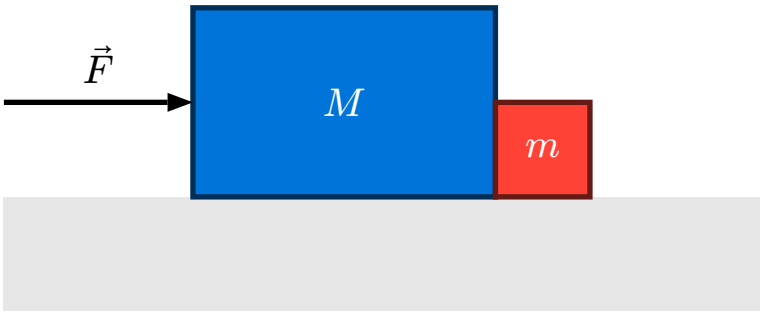
Anchor transformations: `move`, `slide`, `x-inter-x`, `x-inter-y`, `y-inter-x`, `y-inter-y`, `rotate`, `x-look-at`, `y-look-from`, `x-look-from`, `y-look-at`, `pivot`, `lerp`.

Object types: `point`, `arrow`, `spring`, `rope`, `rect`, `circle`, `incline`, `polygon`.

Object transformations: `slide`, `move`, `rotate`, `match`, `stick`.

# 6 Examples

```
let A = rect(50*1.6, 50)
let B = place(rect(25,25)("bl"), A("br"))
let F = place(arrow((0,0), 50, angle: 0deg)("end"), A("l"))
let floor = move(place(rect(200, 30)("t"), A("b")), 10, 0)

draw(floor, fill: luma(90%), stroke: none)
draw(A, fill: blue, stroke: 2pt + blue.darken(60%))
draw(B, fill: red, stroke: 2pt + red.darken(60%))
draw(point(A("c")), label: text(fill: white, $M$), fill:
white)
draw(point(B("c")), label: text(fill: white, $m$), fill:
white, align: center, ly: 1.5)
draw(F, stroke: 2pt)
draw(point(F("c"), rot: false), align: bottom, label:
math.arrow($F$), ly: 5)
```




# Index