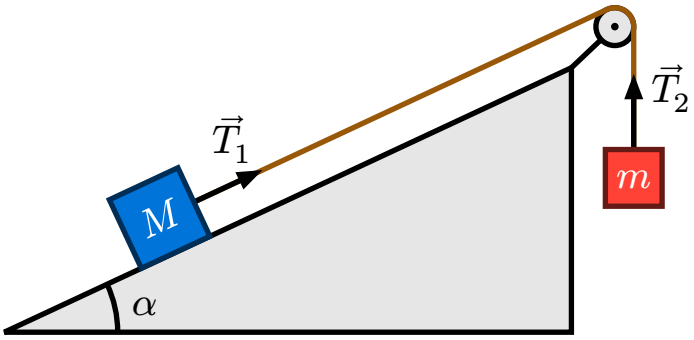
patatrac

|pata'trak|



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

1 Introduction

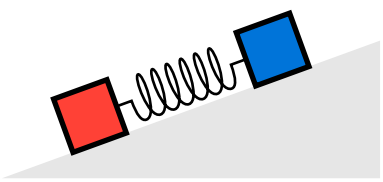
This Typst package provides help with the typesetting of physics diagrams depicting classical mechanical systems. The goal:

drawing beautiful physics diagrams without trigonometry.

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

2 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.



2.1 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 100×20 rectangle. The output of `draw` is a cetz-element that is then picked up by the canvas and printed.

2.2 Introducing anchors

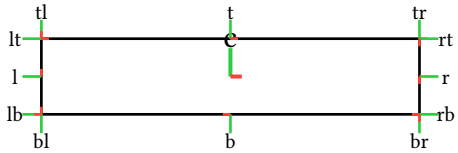
Every object carries with it a set of anchors. Every anchor is a point in space with a specified orientation. As anticipated, objects are functions. In particular, if you call an object on the string `"anchors"`, a complete dictionary of all its anchors is returned. For example, `floor("anchors")` gives

```
(
  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),
)
```

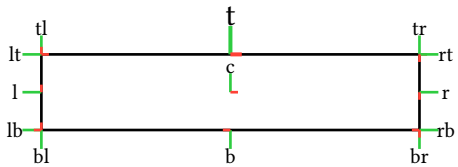
As a general rule of thumb, anchors are placed both at the vertices and at the centers of the sides of the objects and their rotations specify the tangent direction at every point. If you use the renderer `patatrac.cetz.debug` you will see exactly where and how the anchors are placed: red corresponds 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"!

2.3 Composition

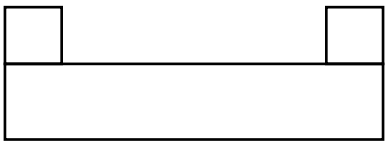
Now, let's add in the two blocks. First of all, we need to place the blocks on top of the floor. To do so we use the `place` function which takes two objects and gives as a result the first object translated such that its active anchor location overlaps with that of the second object.

```
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.

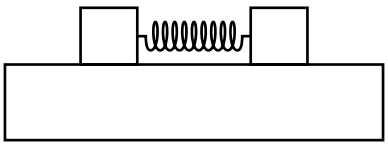
```
let floor = rect(100, 20)
let A = rect(15, 15)
let B = rect(15, 15)

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)
```

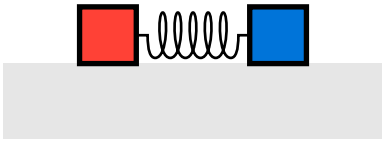


2.4 Styling

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.

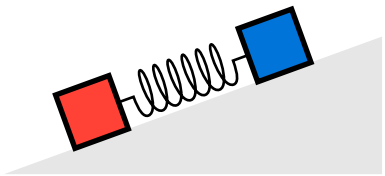
```
// ...

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 (well, drawing the spring is not a piece of cake but bare with me). 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 // ...
```



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.

2.5 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 3 to understand better.

3 Core system

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

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

Let's define them one by one.

3.1 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.

3.2 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
)
```

3.3 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) => { ... }
))
```

4 Some interesting object types

We will now spend a few words to describe how to work with a few types of objects that deserve a special treatment in this discussion. To be clear, `patatrack` treats these like all other object types.

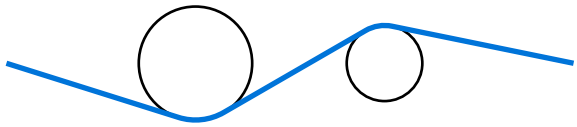
4.1 Ropes

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:

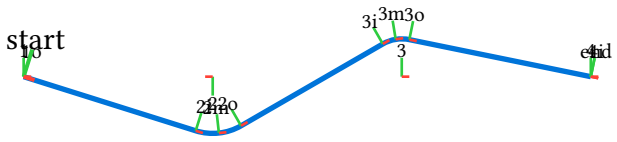
> *ropes are one dimensional strings that wrap around anchors and circles.*

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 C1 = circle(15)
let C2 = place(circle(10), (50, 0))
let R = rope((-50, 0), C1("b"), C2("t"), (+100, 0))
draw(C1, C2)
draw(R, stroke: 2pt + blue)
```



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),
```

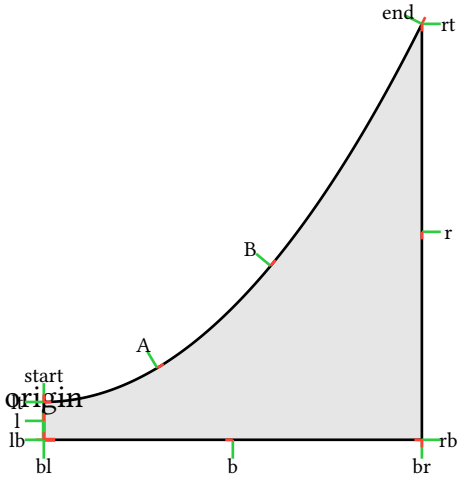
```
  "2": (x: 0, y: 0, rot: 180deg),
  "2i": (x: -4.5, y: -14.309088021254185, rot: 342.54deg),
  "2m": (
    x: 1.638519763245789,
    y: -14.910239870151418,
    rot: 6.27deg,
  ),
  "2o": (
    x: 7.500000000000002,
    y: -12.990381056766578,
    rot: 30deg,
  ),
  "3": (x: 50.0, y: 0.0, rot: 0deg),
  "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),
)
```

Really, there isn't anything more to say about ropes: they just work.

4.2 Terrains

Terrains are objects that you can use to create arbitrarily shaped surfaces. All you need to create a `terrain` is a function describing the profile and its domain, expressed as a tuple `(min, max)`.

```
let ground = terrain(
  x => 0.1 + x*x, (0, 1),
  scale: 100, A: 30%, B: 0.6,
)
draw(ground, fill: luma(90%))
debug(ground)
```

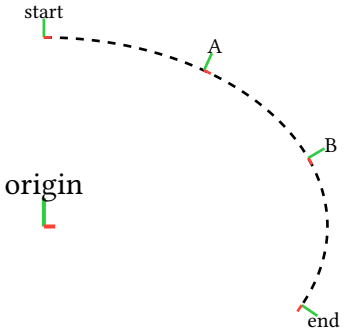


Here we are also specifying two points A and B, by giving their position on the specified range either as number or ratio, which are automatically added as anchors tangent to the surface.

4.3 Trajectories

Trajectories are objects that you can use to create arbitrarily shaped curves. All you need to create a `trajectory` is a function parametrizeing 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,
)
draw(motion, stroke: (dash: "dashed", thickness: 1pt))
debug(motion)
```



Here we are also specifying two points A and B, by giving their position on the specified range either as number or ratio, which are automatically added as anchors tangent to the curve.

5 Useful lists

In this section you'll find a few useful lists. This lists are generated semi-automatically and errors are possible; let me know if you find any.

5.1 Renderers

This is the complete list of available renderers

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

5.2 Objects

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

- `arrow(start, length, angle: none)`
- `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)`
- `match(obj, target, x: true, y: true, rot: true)`
- `move(obj, dx, dy, rot: 0deg)`
- `object(obj-type, active, anchors, data: none)`
- `rotate(obj, angle, ref: none)`
- `slide(obj, dx, dy, rot: none)`
- `stick(obj, target)`
- `transform(obj, func)`

5.3 Anchors

Under the namespace `patatrac.anchors` you can find

- `anchor(x, y, rot)`
- `distance(anchor1, anchor2)`
- `lerp(anchor1, anchor2, by, rot: true)`
- `move(anchor1, dx, dy)`
- `pivot(target, origin, angle, rot: true)`
- `rotate(target, angle)`
- `slide(target, dx, dy, rot: none)`
- `term-by-term-difference(anchor1, anchor2)`
- `term-by-term-sum(anchor1, anchor2)`
- `to-anchor(thing, panic: true)`
- `x-inter-x(anchor1, anchor2)`
- `x-inter-y(anchor1, anchor2)`
- `x-look-at(anchor1, anchor2)`
- `x-look-from(anchor1, anchor2)`
- `y-inter-x(anchor1, anchor2)`
- `y-inter-y(anchor1, anchor2)`
- `y-look-at(anchor1, anchor2)`
- `y-look-from(anchor1, anchor2)`

Index