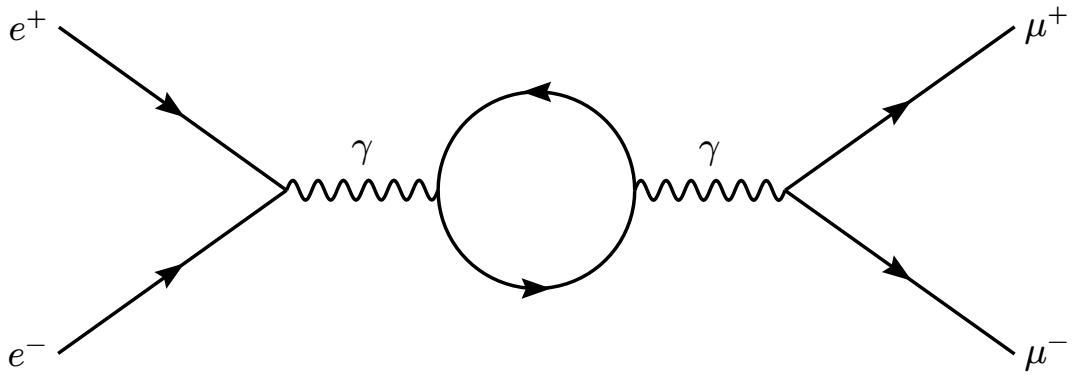
inknertia

v0.1.0 · 8th December 2025

by Pau López Oliver



A **typst** package for drawing physics diagrams: Feynman diagrams, Newtonian mechanics, etc.

Manual & Documentation

Contents

1. Introduction

The `inknertia` (from "ink" + "inertia") package provides tools for drawing various physics diagrams in Typst. It is just a wrapper for the CetZ package, the aim is to provide a set of more *user-friendly* functions to draw common diagrams in physics. The main objective of this package is to fill the void left by the lack of a Feynman diagram package for Typst. However, there are different modules that contain several helper function that may be useful for other purposes.

Currently, the available modules are:

- `feynman`: Module to make Feynman diagrams.
- `newtonian`: Tools for drawing free-body diagrams for classical mechanics.
- `spacetime`: Module to draw two-dimensional spacetime diagrams in special relativity.

The current implementation is half-baked, it is still in development and there are many features that are not yet implemented. We aim to add support for several other diagrams, like geometric optics diagrams. Contributions are more than welcome!

1.1. Getting Started

One can import the entirety of the package with:

```
#import "@preview/inknertia:0.1.0": *
```

2. Feynman Diagrams

2.1. Loading the package

In order to get started with Feynman diagrams, one can import the `feynman` module and all its helper functions with:

```
#import "@preview/inknertia:0.1.0": feynman
#import feynman: *
```

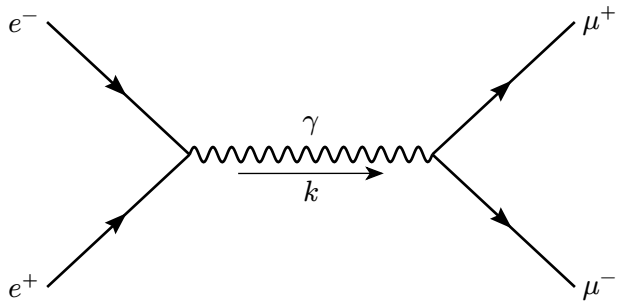
The main advantage over the CetZ package is that it provides a more user-friendly interface to draw Feynman diagrams, since the position of the vertices is automatically computed using a spring-electrical model. This has taken a lot of the work carried out by the author of the LaTeX `TikZ-Feynman` package by Joshua Ellis [1]. The aim of the Feyman module is to provide feature-parity with the LaTeX package even if the syntax is different.

2.2. A simple diagram

Here is a simple example of a diagram for the $e^-e^+ \to \mu^+\mu^-$ interaction where we only specify connectivity: the position of the vertices and length of the propagators is computed automatically.

```
#feynman(
  (
    vertex("i1", label: $e^+$),
    vertex("i2", label: $e^-$),
    vertex("a"),
    vertex("b"),
```

```
    vertex("f1", label: $mu^-$),
    vertex("f2", label: $mu^+$),
    edge("i1", "a", type: "fermion"),
    edge("i2", "a", type: "fermion"),
    edge("a", "b", type: "photon", label: $gamma$, momentum: $k$),
    ..cross(edge("b", "f1", type: "fermion"), edge("b", "f2", type: "fermion")),
  ),
)
```



Let's go through this example line by line:

- **Line 1**: The `#feynman` function is the main entry point. It takes a list of diagram elements (vertices and edges) as its primary argument.
- **Lines 2-8**: We declare the vertices of the graph. `vertex("i1", ...)` creates a node with ID "i1". Since we don't provide a coordinate, the layout engine will determine it automatically using a spring-electrical model. We also assign labels (like e^+) to the external particles. Note that `a` and `b` are internal vertices.
- **Lines 9-11**: We connect the vertices with edges. `edge("i1", "a", type: "fermion")` draws a fermion line (straight line with an arrow) from `i1` to `a`. `edge("a", "b", type: "photon")` draws a wavy photon line between the internal vertices. Notice that we added `momentum: $k$` to the photon edge to display a momentum arrow next to the propagator.
- **Line 12**: We connect the final edges `b`→`f1` and `b`→ `f2`.

2.3. Layout Configuration

You can control the layout of the diagram using several parameters in the `#feynman` function.

2.3.1. Orientation

You can control the general layout direction using the `orientation` parameter:

- `orientation: "horizontal"` (default): uses the `left` and `right` parameters to pin nodes to the left and right borders of the diagram, respectively.
- `orientation: "vertical"`: uses the `bottom` and `top` parameters to pin nodes to the bottom and top borders.

2.3.2. Debug Mode

You can enable debug mode to visualize the vertex IDs during development:

```
#feynman(debug: true, ...)
```

When `debug: true`, each vertex displays its ID as a small blue label. This is useful for:

- Troubleshooting layout issues

- Identifying which vertex is which when building complex diagrams
- Verifying edge connections

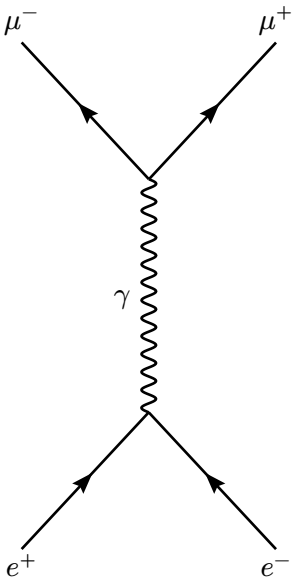
2.3.3. Size

You can control the overall size of the diagram using the `length` parameter. This parameter is passed to `CetZ` and sets the physical length corresponding to one unit in the coordinate system used by the layout engine. The default value is `0.5cm`.

By adjusting `length`, you can scale the entire diagram up or down without changing the relative positions of the vertices.

In the following example, we illustrate both of these concepts. We set `orientation: "vertical"` and anchor the node `i2` to the `top` to rotate the layout. Additionally, we slightly reduce the `length` to `0.48cm` to scale down the entire diagram.

```
#feynman(
  orientation: "vertical",
  top: "i2",
  length: 0.48cm,
  (
    vertex("i1", label: $e^+$),
    vertex("i2", label: $e^-$),
    vertex("a"),
    vertex("b"),
    vertex("f1", label: $mu^-$),
    vertex("f2", label: $mu^+$),
    edge("i1", "a", type: "fermion"),
    edge("i2", "a", type: "fermion"),
   edge("a", "b", type: "photon", label:
$gamma$),
    edge("b", "f1", type: "fermion"),
    edge("b", "f2", type: "fermion"),
  ),
)
```



2.3.4. Layout Algorithms

The package supports three algorithms for automatically positioning vertices. You can select the algorithm using the `layout` parameter in the `#feynman` function.

2.3.4.1. Spring Layout (Default)

```
#feynman(layout: "spring", ...)
```

Uses a force-directed graph drawing algorithm (spring-electrical model). Nodes repel each other while edges act as springs pulling connected nodes together. After many iterations, the system reaches an equilibrium that tends to place connected nodes close together while keeping unconnected nodes apart.

Best for: General-purpose diagrams, s-channel and t-channel scattering diagrams, diagrams without a clear hierarchical structure.

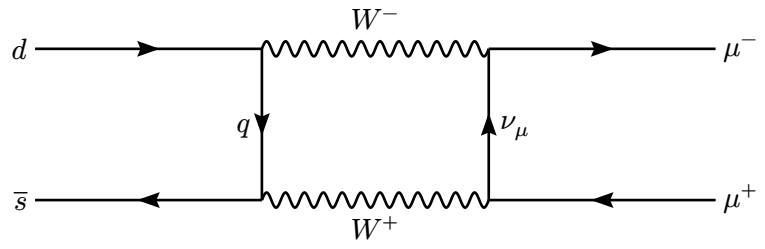
2.3.4.2. Layered Layout

```
#feynman(layout: "layered", ...)
```

Uses a layered graph drawing algorithm (Sugiyama-style). Nodes are assigned to discrete layers based on their topological order, then positioned to minimize edge crossings. The first node defined appears at the top of each layer.

Best for: Box diagrams, ladder diagrams, and any diagram with a clear "flow" or "time" direction where you want edges to predominantly go left-to-right.

```
#feynman(
  layout: "layered",
  orientation: "horizontal",
  (
    vertex("i1", label: $d$),
    vertex("a"),
    vertex("b"),
    vertex("i2", label: $overline(s)$),
    vertex("c"),
    vertex("d"),
    vertex("f1", label: $mu^-$),
    vertex("f2", label: $mu^+$),
    // Top line
    edge("i1", "a", type: "fermion"),
    edge("a", "b", type: "photon", label: $W^-$, label_anchor: "south"),
    edge("b", "f1", type: "fermion"),
    // Bottom line
    edge("i2", "c", type: "antifermion"),
    edge("c", "d", type: "photon", label: $W^+$, label_anchor: "north"),
    edge("d", "f2", type: "antifermion"),
    // Internal vertical lines
    edge("a", "c", type: "fermion", label: $q$, label_anchor: "east"),
    edge("b", "d", type: "antifermion", label: $nu_mu$, label_anchor: "west"),
  ),
)
```



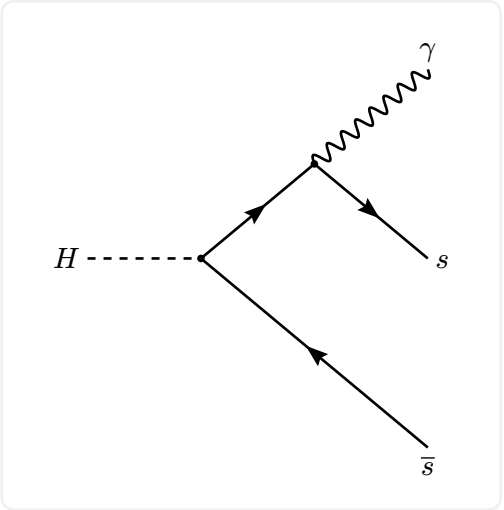
2.3.4.3. Tree Layout

```
#feynman(layout: "tree", ...)
```

Uses the Reingold-Tilford tree drawing algorithm. Parent nodes are automatically centered over their children, and subtrees are drawn without overlapping. Leaf nodes at shallower depths are automatically extended to the maximum tree depth with proportionally scaled positions, maintaining consistent branch angles.

Best for: Decay cascades, hierarchical structures, and diagrams where particles branch into multiple products.

```
#feynman(
  layout: "tree",
  orientation: "horizontal",
  (
    vertex("H", label: $H$),
    vertex("a", shape: "dot"),
    vertex("b", shape: "dot"),
    vertex("sbar", label: $overline(s)$),
    vertex("s", label: $s$),
    vertex("gamma", label: $gamma$),
    // H -> vertex
    edge("H", "a", type: "scalar"),
    // // vertex -> final states
    edge("a", "sbar", type: "antifermion"),
    edge("b", "s", type: "fermion"),
    edge("b", "gamma", type: "photon"),
    edge("a", "b", type: "fermion"),
  ),
)
```



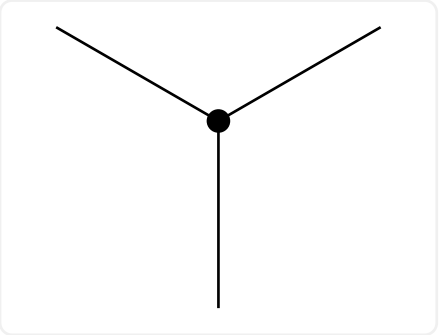
2.4. Vertices

The library supports several standard vertex shapes. You can customize their fill, stroke, and label.

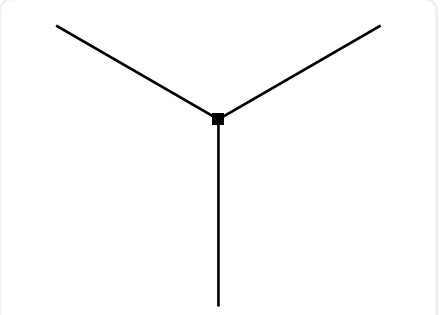
2.4.1. Vertex shapes

The different vertex shapes available in the library are `none` (default), `dot`, `crossed-dot`, `square`, and `blob`.

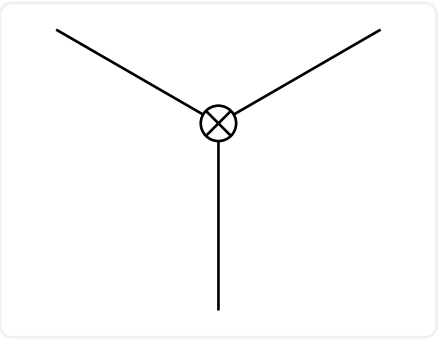
```
#feynman(orientation: "vertical", top: "e1", bottom:
"c", length: 0.52cm, (
  vertex("e1"), vertex("e2"), vertex("e3"),
  vertex("c", shape: "dot", size: 0.3),
  edge("c", "e1", type: "line"),
  edge("c", "e2", type: "line"),
  edge("c", "e3", type: "line"),
  )
)
```



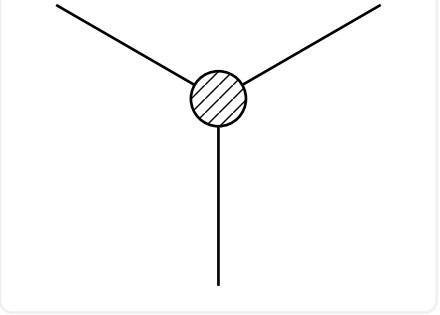
```
#feynman(orientation: "vertical", top: "e1", bottom:
"c", length: 0.52cm, (
  vertex("e1"), vertex("e2"), vertex("e3"),
  vertex("c", shape: "square"),
  edge("c", "e1", type: "line"),
  edge("c", "e2", type: "line"),
  edge("c", "e3", type: "line"),
  )
)
```



```
#feynman(orientation: "vertical", top: "e1", bottom:
"c", length: 0.52cm, (
  vertex("e1"), vertex("e2"), vertex("e3"),
  vertex("c", shape: "crossed-dot"),
  edge("c", "e1", type: "line"),
  edge("c", "e2", type: "line"),
  edge("c", "e3", type: "line"),
  )
)
```



```
#feynman(orientation: "vertical", top: "e1", bottom:
"c", length: 0.52cm, (
  vertex("e1"), vertex("e2"), vertex("e3"),
  vertex("c", shape: "blob", size: 0.7),
  edge("c", "e1", type: "line"),
  edge("c", "e2", type: "line"),
  edge("c", "e3", type: "line"),
  )
)
```



You can also customize individual attributes like `hatch_spacing` for blobs or the fill to make something like an empty dot:

```
#feynman((
  vertex("a", shape: "dot", fill: orange, stroke: red),
  vertex("a2", shape: "dot", fill: white, stroke: black, size: 0.15cm),
  vertex("b", shape: "square", fill: blue),
  vertex("c", shape: "crossed-dot", stroke: green),
  vertex("d", shape: "blob", stroke: purple, hatch_spacing: 0.2cm, size: 0.5cm),
  edge("a", "a2", type: "scalar"),
  edge("a2", "b", type: "scalar"),
  edge("b", "c", type: "scalar"),
  edge("c", "d", type: "scalar"),
))
```

The available named parameters for the vertices are:

- `label`: The content to display near the vertex (e.g., `$e^-$`).
- `label_anchor`: Where to anchor the label relative to the vertex (default: `"auto"`).
- `label_padding`: Distance between the vertex and the label (default: `0.2`).
- `label_fill`: Text color of the label (default: `black`).
- `shape`: The visual shape of the vertex (`"dot"`, `"square"`, `"crossed-dot"`, `"blob"`).
- `size`: The radius or size of the vertex shape.
- `fill`: The fill color of the vertex shape.
- `stroke`: The stroke color/style of the vertex shape.

- `hatch_spacing`: Spacing for the hatching pattern if shape is `"blob"`.

2.5. Edges

The `edge` function connects two vertices and draws a propagator between them.

2.5.1. Edge Types

The `type` parameter controls the physics representation of the line:

- **Fermions**: `fermion`, `antifermion`, `majorana`, `anti-majorana`
- **Bosons**: `photon`, `boson` (alias for photon), `gluon`
- **Charged Bosons**: `charged-boson`, `anti-charged-boson`
- **Scalars**: `scalar` (dashed), `charged-scalar`, `anti-charged-scalar`
- **Ghosts**: `ghost` (dotted)

2.5.2. Styling Parameters

You can customize the appearance with the following named arguments:

- `color`: The color of the edge (default: `black`).
- `stroke_width`: Thickness of the line (default: `1pt`).
- `label`: Content to display on the edge.
- `label_dist`: Distance of the label from the edge (default: `0.3`).
- `label_anchor`: Position of the label (default: `"auto"`).
- `label_fill`: Color of the label text.
- `arrow_size`: Size of the arrows (default: `0.6`).
- `amplitude`: Amplitude of the sine/coil wave (default: `0.2`).
- `period`: Period of the wave (default: `0.5`).
- `momentum`: Content for a momentum arrow alongside the edge.
- `momentum_dist`: Distance of the momentum arrow (default: `0.5`).
- `momentum_flip`: If `true`, flips the momentum arrow to the other side.
- `momentum_stroke`: Color of the momentum arrow.
- `momentum_fill`: Color of the momentum text.

2.5.3. Examples

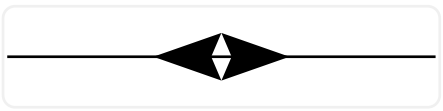
Here are examples of the various edge types:

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "fermion")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "antifermion")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "majorana")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "anti-majorana")))
```



```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "photon")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "boson")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "charged-boson")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "anti-charged-boson")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "scalar")))
```



```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "charged-scalar")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "anti-charged-scalar")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "gluon")))
```

```
#feynman(length: 1.5cm, (vertex("a"), vertex("b"),
edge("a", "b", type: "ghost")))
```


2.6. Crossings

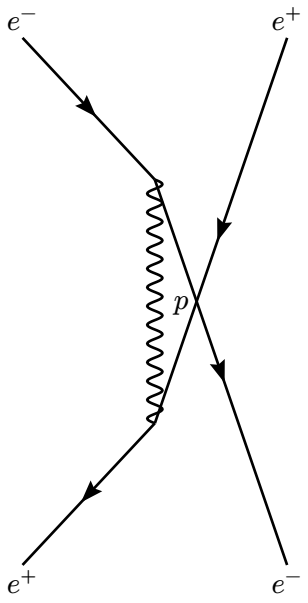
You can construct topologically different Feynman diagrams by crossing any two edges of the diagram using the `cross` function.

This function takes two edge objects as arguments. It maintains their original start and end points for the purpose of the layout algorithm (so the nodes stay in their logical positions), but swaps their end

points when drawing the lines. This creates a visual "crossing" effect without affecting the underlying graph structure used for positioning.

Note that `cross` returns an array of two edges, so you must use the spread operator (`..cross(...)`) when including them in the diagram's item list.

```
#feynman(orientation: "vertical", (
  vertex("i1", label: $e^-$),
  vertex("i2", label: $e^+$),
  vertex("a"),
  vertex("b"),
  vertex("f1", label: $e^+$),
  vertex("f2", label: $e^-$),
  edge("i1", "a", type: "fermion"),
  edge("i2", "b", type: "antifermion"),
  edge("a", "b", type: "photon", label:
$p$),
  ..cross(
    // Cross outgoing fermions
    edge("a", "f1", type: "fermion"),
   edge("b", "f2", type: "antifermion"),
  ),
))
```



2.7. Loops

The library includes a helper function `loop` to easily create circular loop diagrams.

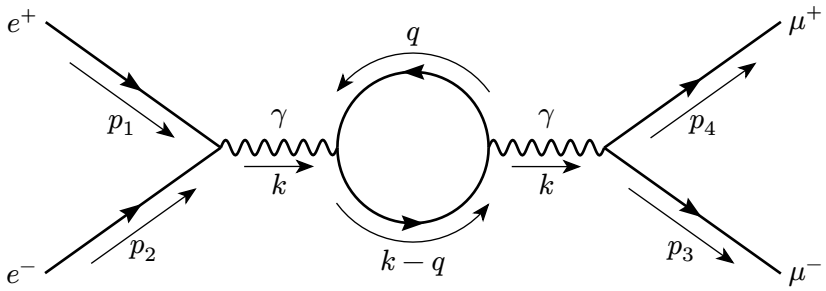
The `loop` function takes a variable number of arguments. Each argument represents a node in the loop and the edge connecting it to the next node. An argument can be:

- A `vertex` object (e.g., `vertex("a")`). In this case, the edge connecting it to the next node will be a standard straight line (unless overridden by the loop mechanics).
- A tuple `(vertex, edge_type_string)` or `(vertex, edge_properties_dictionary)`. This allows you to specify the type (e.g., `"photon"`, `"gluon"`) or other properties of the edge connecting this vertex to the next one in the circular sequence.

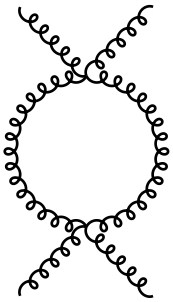
The function automatically positions the vertices in a circle around a `center` (default: `(0,0)`) with a specified `radius` (default: `2.0`) and creates curved edges between them to form a closed loop. The result is a list of diagram items (vertices and edges) that must be unpacked using the spread operator (`..`).

```
#feynman(
  (
    vertex("i1", label: $e^+$),
    vertex("i2", label: $e^-$),
    vertex("f1", label: $mu^-$),
    vertex("f2", label: $mu^+$),
    vertex("a"),
    edge("i1", "a", type: "fermion", momentum: $p_1$),
    edge("i2", "a", type: "fermion", momentum: $p_2$),
    edge("a", "b", type: "photon", label: $gamma$, momentum: $k$),
    // Loop between b and c
    loop(
```

```
      (vertex("c"), (type: "fermion", momentum: $q$)),
      (vertex("b"), (type: "fermion", momentum: $k - q$)),
    ),
    vertex("d"),
    edge("c", "d", type: "photon", label: $gamma$, momentum: $k$),
    edge("d", "f1", type: "fermion", momentum: $p_3$),
    edge("d", "f2", type: "fermion", momentum: $p_4$),
  ),
)
```



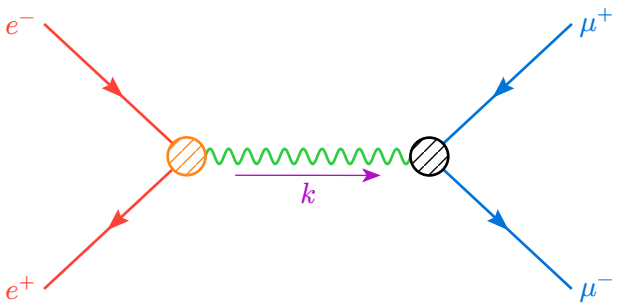

```
#feynman(orientation: "vertical", (
       vertex("i1"),   vertex("i2"),   vertex("f1"),
vertex("f2"),
  edge("i1", "a", type: "gluon"),
  edge("i2", "a", type: "gluon"),
  loop(
    (vertex("a"), "gluon"),
    (vertex("b"), "gluon")),
  edge("b", "f1", type: "gluon"),
  edge("b", "f2", type: "gluon"),
))
```



2.7.1. Color and Customization

```
#feynman(
  (
    vertex("i1", label: $e^-$, fill: red, label_fill: red),
    vertex("i2", label: $e^+$, fill: red, label_fill: red),
    vertex("a", shape: "blob", fill: yellow, stroke: orange),
    vertex("b", shape: "blob"),
    vertex("f1", label: $mu^+$, fill: blue, label_fill: blue),
    vertex("f2", label: $mu^-$, fill: blue, label_fill: blue),
    edge("i1", "a", color: red),
    edge("a", "i2", color: red),
   edge("a", "b", type: "photon", color: green, momentum: $k$, momentum_stroke: purple,
momentum_fill: purple),
    edge("f1", "b", color: blue),
    edge("b", "f2", color: blue),
  ),
  left: "i1",
```

```
  right: "f1",
)
```

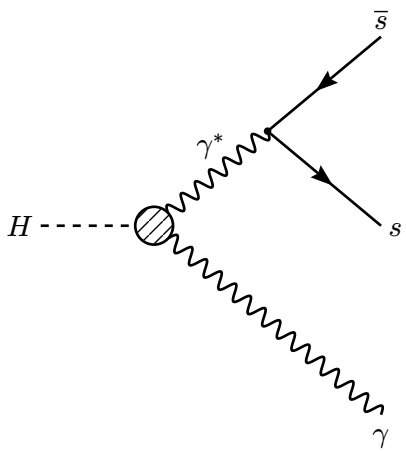


2.8. Examples

Below is a gallery of Feynman diagrams. They have been created using different layouts ir order to show the capabilities of the library.

2.8.1. Higgs Decay Diagrams

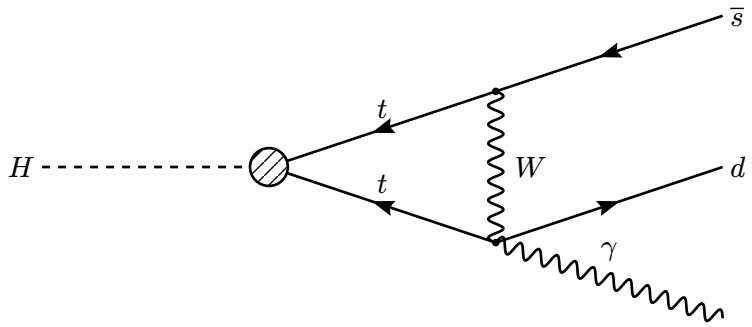
```
#feynman(layout: "tree", orientation: "horizontal", (
  vertex("H", label: $H$),
  vertex("a", shape: "blob"),
  vertex("b", shape: "dot"),
  vertex("gamma", label: $gamma$),
  vertex("s", label: $s$),
  vertex("sbar", label: $overline(s)$),
  edge("H", "a", type: "scalar"),
  edge("a", "gamma", type: "photon"),
  edge("b", "s", type: "fermion"),
  edge("b", "sbar", type: "antifermion"),
  edge("a", "b", type: "photon", label: $gamma^(*)$),
))
```



A diagram for the $H \to \bar{s}d$ `gamma` decay is best drawn using the `layered` layout.

```
#feynman(layout: "layered", debug: false, orientation: "horizontal", (
  vertex("H", label: $H$),
  vertex("a", shape: "blob"),
  vertex("b", shape: "dot"),
  vertex("c", shape: "dot"),
  vertex("f1", label: $overline(s)$),
  vertex("f2", label: $d$),
  vertex("f3"),
  edge("H", "a", type: "scalar"),
  edge("a", "c", type: "antifermion", label: $t$),
  edge("b", "f1", type: "antifermion"),
  edge("c", "f2", type: "fermion"),
  edge("a", "b", type: "antifermion", label: $t$),
```

```
  edge("b", "c", type: "boson", label: $W$),
  edge("c", "f3", type: "boson", label: $gamma$),
))
```



3. Classical Mechanics

The `newtonian` module provides helper functions for drawing free body diagrams, including springs, pulleys, ropes, vectors, walls, and coordinate axes. Import it with:

```
#import "package/newtonian.typ": *
```

All functions are designed to be used inside a `cetz.canvas` block.

3.1. Spring

Draws a coiled spring between two points.

```
#spring(start, end, R, coils, ...)
```

Parameters:

- `start`: Starting point `(x, y)`
- `end`: Ending point `(x, y)`
- `R`: Radius of the coils
- `coils`: Number of coils
- `starthook`, `endhook`: Percentage of length for straight hooks at each end (default: `0%`)
- `startcircle`, `endcircle`: Draw a small circle at the endpoints (default: `false`)
- `color`: Color of the spring (default: `black`)
- `thickness`: Stroke thickness (default: `1pt`)
- `name`: CetZ element name for referencing

```
#cetz.canvas({
  import cetz.draw: *
    spring((0,  0),  (10,  0),  0.5,  10,
starthook: 0%, endhook: 5%, startcircle:
true, color: gray)
})
```

3.2. Wall

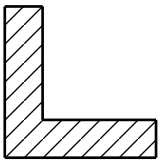
Draws a filled polygon representing a wall or solid surface.

```
#wall(points, fill_paint: hatch(), stroke_style: none, sides: "all")
```

Parameters:

- `points`: Array of `(x, y)` coordinates forming the polygon
- `fill_paint`: Fill color (default: `hatch()`)
- `stroke_style`: Stroke for edges, or `none` for no stroke (default: `none`)
- `sides`: Which sides to stroke—`"all"` or an array of side indices (default: `"all"`)

```
#cetz.canvas({
  wall(
    ((0, 0), (0, 2), (0.5, 2), (0.5, 0.5), (2, 0.5),
(2, 0)),
    stroke_style: 1pt + black,
  )
})
```



3.3. Pulley

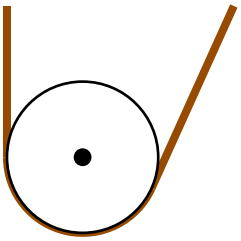
Draws a pulley wheel, optionally with a rope threaded through it.

```
#pulley(pos, R: 1, fill_paint: white, stroke_style: 1pt + black, ...)
```

Parameters:

- `pos`: Center position of the pulley `(x, y)`
- `R`: Radius of the pulley wheel (default: `1`)
- `fill_paint`: Fill color (default: `white`)
- `stroke_style`: Stroke for the wheel (default: `1pt + black`)
- `rope_start`, `rope_end`: If both are provided, a rope is drawn from `rope_start` around the pulley to `rope_end`
- `rope_stroke`: Stroke style for the rope (default: `3pt + rgb("964B00")`)

```
#cetz.canvas({
  import cetz.draw: *
  pulley((0, 0), rope_start: (-1, 2), rope_end: (2, 2))
})
```



3.4. Rope

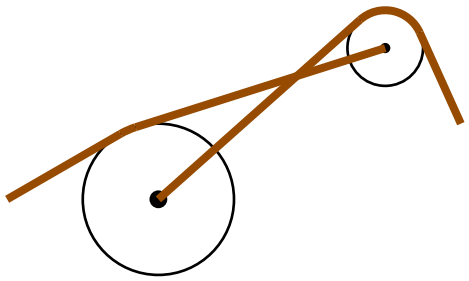
Draws a rope that wraps around a pulley. This is automatically called by `pulley()` when `rope_start` and `rope_end` are specified, but can also be used standalone.

```
#rope(pulley, start, end, radius: 1, stroke_style: 3pt + rgb("964B00"))
```

Parameters:

- `pulley`: Center position of the pulley `(x, y)`
- `start`: Start point of the rope `(x, y)`
- `end`: End point of the rope `(x, y)`
- `radius`: Radius of the pulley (default: `1`)
- `stroke_style`: Stroke style (default: `3pt + rgb("964B00")`)

```
#cetz.canvas({
  let p1 = (0, 0)
  let p2 = (3, 2)

  pulley(p1)
  pulley(p2, R: 0.5)

  // Rope connecting the tops of the pulleys
  rope(p1, (-2, 0), p2, radius: 1)
  rope(p2, p1, (4, 1), radius: 0.5)
})
```



3.5. Axes

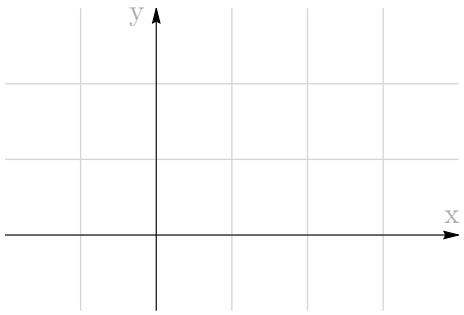
Draws Cartesian coordinate axes with optional grid.

```
#axes(xmin: -10, xmax: 10, ymin: -10, ymax: 10, xlabel: "x", ylabel: "y", ...)
```

Parameters:

- `xmin`, `xmax`: Horizontal extent of the x-axis
- `ymin`, `ymax`: Vertical extent of the y-axis
- `xlabel`, `ylabel`: Labels for the axes (default: `"x"`, `"y"`)
- `color`: Color for the axes and labels (default: `gray`)
- `grid_stroke`: If specified, draws a grid with this stroke style (default: `none`)
- `grid_step`: Grid spacing (default: `1`)

```
#cetz.canvas({
  axes(
    xmin: -2,
    xmax: 4,
    ymin: -1,
    ymax: 3,
    grid_stroke: 0.5pt + gray.lighten(50%),
    grid_step: 1,
  )
})
```



3.6. Vector

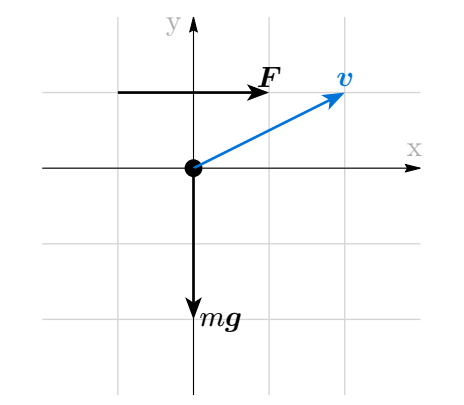
Draws an arrow with an optional label.

```
#vector(start, end, label: none, stroke_style: 1pt + black, fill_paint: black, anchor: "auto", padding: 0.2em)
```

Parameters:

- `start`: Starting point `(x, y)`
- `end`: Ending point `(x, y)` where the arrowhead is drawn
- `label`: Optional label content displayed at the tip
- `stroke_style`: Stroke for the arrow shaft (default: `1pt + black`)
- `fill_paint`: Fill color for the arrowhead and label (default: `black`)
- `anchor`: Anchor position for the label—`"auto"` places it at `"south"` (default: `"auto"`)
- `padding`: Padding around the label (default: `0.2em`)

```
#cetz.canvas({
  import cetz.draw: *

   axes(xmin: -2, xmax: 3, ymin: -3, ymax: 2, grid_stroke: 0.5pt + gray.lighten(50%))

  circle((0, 0), radius: 0.1, fill: black)
    vector((0, 0), (2, 1), label: $bold(v)$, fill_paint: blue, stroke_style: 1pt + blue)
   vector((0, 0), (0, -2), label: $m bold(g)$, anchor: "west")
  vector((-1, 1), (1, 1), label: $bold(F)$)
})
```



3.7. Curved Arrow

Draws a curved arc arrow, useful for angular velocity or rotation indicators.

```
#curved_arrow(center, radius: 1, start_angle: 0deg, end_angle: 90deg, direction: "auto", ...)
```

Parameters:

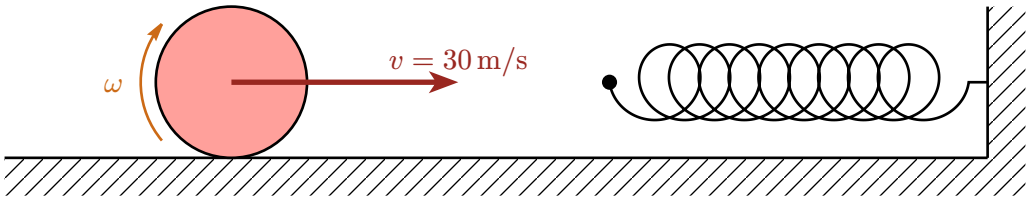
- `center`: Center point of the arc `(x, y)`

- `radius`: Radius of the arc (default: `1`)
- `start_angle`: Starting angle (default: `0deg`)
- `end_angle`: Ending angle (default: `90deg`)
- `direction`: Arc direction—`"auto"`, `"cw"`, or `"ccw"` (default: `"auto"`)
- `color`: Color of the arrow and label (default: `black`)
- `thickness`: Stroke thickness (default: `1pt`)
- `label`: Optional label content displayed along the arc
- `label_radius`: Distance from center for label (default: `auto` = 1.3 × radius)

3.8. Examples

3.8.1. Spring System

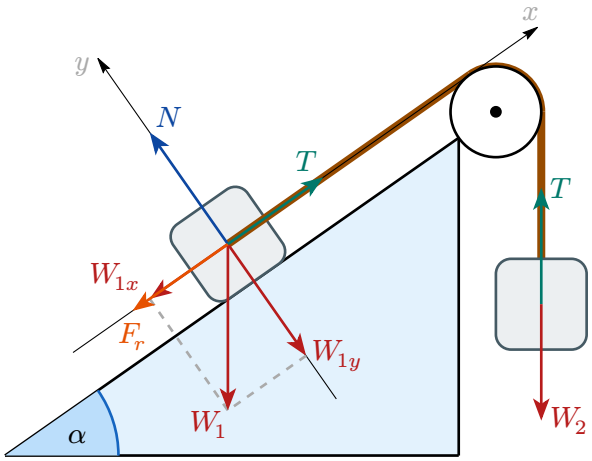
```
#cetz.canvas(length: 1cm, {
  import cetz.draw: *

  wall(((-3, 0), (10, 0), (10, 2), (10.5, 2), (10.5, -0.5), (-3, -0.5)), stroke_style:
1pt + black, sides: (0, 1))
  spring((5, 1), (10, 1), 0.5, 10, startcircle: true, endhook: 5%)
  circle((0, 1), radius: 1, fill: red.lighten(50%))
   curved_arrow((0, 1), radius: 1.2, start_angle: 220deg, end_angle: 140deg, color:
orange.darken(20%), label: $omega$)
  vector((0, 1), (3, 1), label: $v = 30 thin "m/s"$, stroke_style: 2pt + red.darken(40%),
fill_paint: red.darken(40%))
})
```



3.8.2. Inclined Plane

This example shows a block on an inclined plane connected to a hanging mass. It illustrates how to use the library to create more involved diagrams, including:

- Rotating the coordinate system to match the slope.
- Decomposing vectors into components.

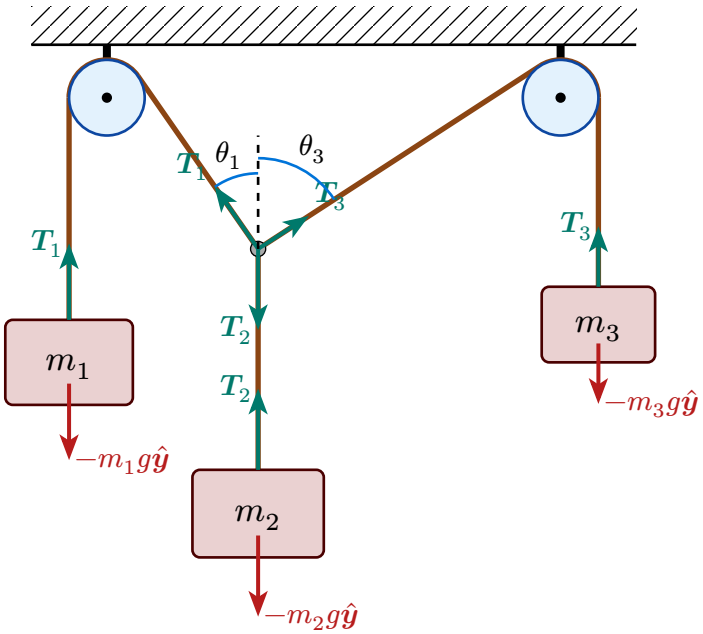


You can check the source code for this diagram in `docs/diagrams/newtonian/inclined_plane.typ`.

3.8.3. Equilibrium of Three Masses

This diagram depicts a system of three masses in static equilibrium, suspended by ropes over two pulleys. It demonstrates:

- Calculating mass values based on angles to ensure physical equilibrium.
- Drawing ropes tangent to pulleys.
- Visualizing forces with vectors originating from a central knot.



4. Spacetime Diagrams

The `spacetime` module enables the creation of Minkowski diagrams for special relativity. It operates in natural units ($c = 1$) so light rays always travel at 45 degrees.

To use it, import the module:

```
#import "@preview/inknertia:0.1.0": spacetime
```

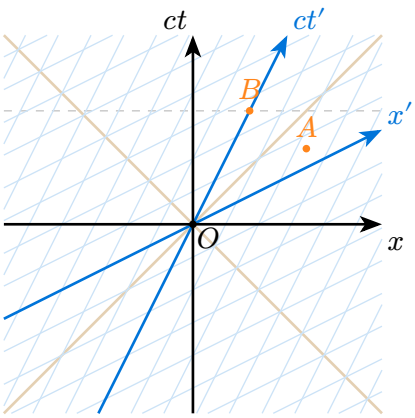
```
#import spacetime: *
```

4.1. A first diagram

The main container is `#spacetime()`. By default, it operates in natural units ($c = 1$), so the vertical axis is labeled t. You can switch to standard units (ct) by setting `natural_units: false`.

You can also enable a background grid using the `grid` parameter, which accepts a stroke style.

```
#let x = 1.5

#spacetime(
  natural_units: false,
  grid_step: 1,
  (
      frame(beta: 0.5, grid_stroke: (paint: blue.lighten(80%), thickness: 0.5pt),
grid_spacing: 1),
    event("O", (0, 0), label: $O$, anchor: "north-west", padding: 0.1),
    event("A", (3, 2), label: $A$, color: orange, anchor: "south", padding: 0.3),
    event("B", (x, 2 * x), label: $B$, color: orange, anchor: "south", padding: 0.3),
    simultaneity(2 * x, color: gray.lighten(30%)),
    lightcone((0, 0)),
  ),
)
```



Bibliography

[1] J. P. Ellis, "Ti k Z-Feynman: Feynman diagrams with Ti k Z," *Computer Physics Communications*, vol. 210, pp. 103–123, Jan. 2017, doi: 10.1016/j.cpc.2016.08.019.