

What is PGF/Tikz?

PGF/Tikz is a pair of languages for generating *vector graphics*. That is, graphics constructed from geometric primitives (points/vertices, line segments, polyhedra, parametric surfaces, etc.), with each primitive assigned a set of properties (colour, line weight, dash patterns, etc.). This is in contrast to *raster graphics*, in which each pixel of the image is stored (e.g. with colour, luminance, transparency values, etc.).

Because vector graphics consist of coordinates/lines/curves, the size of a representation on a screen is independent of the size of the object – you can zoom in to a vector image arbitrarily close and it will remain crisp.

Furthermore, the parameters of vector objects can be later modified, so that moving, scaling, rotating, filling, etc. does not degrade the quality of a vector image. In contrast, repeated modifications to a raster image will continually degrade the image quality as elements are snapped to a pixel grid between each edit.

Getting Started

A Tikz diagram is built within the `tikzpicture` environment. Primitives are specified and drawn using Tikz commands within the environment, each terminated with a semicolon (;).

Here is an example of some Tikz code, and its output:

Specifying Points

There are many ways to specify *points* or *coordinates* in Tikz. You can declare your coordinate system explicitly (this will be more relevant later) with the syntax

```
([coordinate system] cs: (system-specific coordinates))
```

but for various common coordinate systems, special implicit syntax is available.

For instance, cartesian coordinate may be specified explicitly using the `canvas` coordinate system:

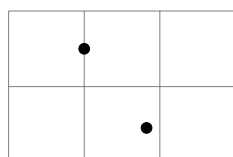
```
(canvas cs:x=2ex,y=5pt)
```

which takes two keys, `x` and `y`, that accept \TeX dimensions. Or, they can be specified implicitly by listing the two dimensions separated by commas in round brackets, as in

```
(2ex, 5pt)
```

This coordinate means “2ex upwards and 5pt to the right of the origin”.

Note that you can also write things like `1em+2cm` in a dimension, since the maths engine is used to evaluate the coordinates.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \filldraw (1cm,1.5cm) circle (2pt);
  \filldraw (2cm-5pt,3ex) circle (2pt);
\end{tikzpicture}
```

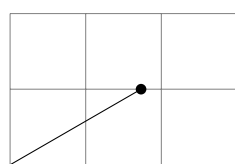
Polar coordinates may be specified using the `canvas polar` system

```
(canvas polar cs:angle=30,radius=2cm)
```

or implicitly by providing the angle and a radius separated by a colon:

```
(30:2cm)
```

This coordinate means “2cm from the origin, 30 degrees counterclockwise from the positive x -axis”.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (30:2cm);

  \fill (30:2cm) circle (2pt);
\end{tikzpicture}
```

There are other coordinate systems, but we’ll only discuss those on request. From now on, we’ll also omit the explicit system declaration unless it is otherwise relevant.

If units are not provided, e.g. $(1, 2)$, then the coordinates are specified in PGF’s internal xy -coordinate system. By default, the unit x -vector points 1cm to the right, and the unit y -vector points 1cm upwards.*

By giving three numbers, as in $(1, 2, 3)$, the coordinates are specified in PGF’s internal xyz -coordinate system. By default, the unit x -vector points 1cm to the right, and the unit y -vector points 1cm upwards, and the unit z -vector points to $(-3.85\text{mm}, -3.85\text{mm})$.

It is also possible to use an anchor (explained later) of an existing shape as in `(my_node.centre)` as a coordinate.

You can add two plus signs before a coordinate to specify a coordinate relative to the previously defined coordinate. For example, $(1, 0) ++ (1, 0) ++ (0, 1)$ specifies the three coordinates $(1, 0)$, then $(2, 0) [= (1, 0) + (1, 0)]$, and $(2, 1) [= (2, 0) + (0, 1)]$.

Instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but doesn’t update the reference coordinate. For example, $(1, 0) + (1, 0) + (0, 1)$ specifies the three coordinates $(1, 0)$, then $(2, 0) [= (1, 0) + (1, 0)]$, and $(1, 1) [= (1, 0) + (0, 1)]$.

Paths

A *path* is a series of curves between points, which need not be connected.

To specify a straight line path between two coordinates, put two dashes between the coordinates. For instance,

```
(0,0) -- (0,1) -- (1,1) -- cycle
```

specifies a triangular path, with the special *cycle* coordinate *smoothly* closing the path. To see the difference between using *cycle* and simply repeating the first coordinate, consider the following:



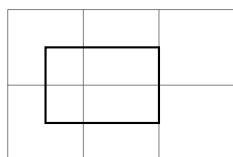
```
\begin{tikzpicture}[line width=5pt]
  \path[draw] (0,0) -- (0,1) -- (1,1) -- cycle;
  \path[draw] (2,0) -- (2,1) -- (3,1) -- (2,0);
\end{tikzpicture}
```

Shapes and Special Paths

There are various commands for generating predefined shapes and patterns. For instance,

```
(a,b) rectangle (c,d)
```

specifies a rectangular box with opposite corners at (a, b) and (c, d) .



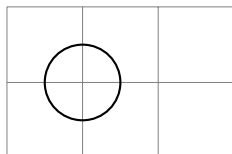
```
\begin{tikzpicture}[line width=5pt]
  \draw[help lines] (0,0) grid (3,2);

  \path[draw] (0.5,0.5) rectangle (2.5,1.5);
\end{tikzpicture}
```

*It is possible to use coordinates like $(1, 2\text{cm})$, which is neither an xy -coordinate nor a canvas coordinate. Roughly speaking, if you have mixed coordinates, Tikz will typecast the coordinate $(X, Y\text{cm})$ to the sum $(X, 0) + (0\text{pt}, Y)$, and

`(a,b) circle (r)`

specifies a circle centred at (a,b) with radius r .



```
\begin{tikzpicture}[line width=5pt]
  \draw[help lines] (0,0) grid (3,2);

  \path[draw] (1,1) circle (0.5);
\end{tikzpicture}
```

Actions

Note that, so far, a path is simply a series of curves and coordinates – we haven't actually told Tikz what to do with them.

Given a path, we may:

- *draw* or *stroke* the path;
- *fill* the path;
- *clip* the path;
- (various other specialised options);
- or do any combination of the above.

To perform these actions, pass them as options to the `\path` command or use one of these aliases:

```
\draw ...      := \path[draw] ...
\fill ...      := \path[fill] ...
\filldraw ...  := \path[fill,draw] ...
etc.
```



```
\begin{tikzpicture}
  \path (0,0) circle (0.5); % does nothing
  \draw (2,0) circle (0.5);
  \fill (4,0) circle (0.5);
\end{tikzpicture}
```

There are many other options you can pass to a draw command, but some important ones include:

- colours;
- opacities;
- stroke width;
- dash patterns;
- arrow tips.


Passing a colour to a `\draw` command will set the global colour in that scope, so all lines, fills, labels, etc. will all use that colour unless overridden locally. (Opacity has a similar discussion.) If any options are unset, they will default to black/full opacity (defaults may also be modified).

More specialised options that only set individual colours/opacities may also be used:

similarly for any mixed dimensions. In particular, $(2+3\text{cm}, 0)$ is *not* the same as $(2\text{cm}+3\text{cm}, 0)$, as it will be typeset to the sum $(2\text{pt}, 0) + (3\text{cm}, 0) = (2\text{pt}+3\text{cm}, 0)$.

```

\begin{tikzpicture}[line width=2pt]
  \draw (2,0) node {$1$} (4,0) node {$2$};



  \filldraw[red] (0,0) circle (0.5);
  \filldraw[draw=blue,fill=orange] (2,0) circle (0.5);
  \filldraw[blue,fill opacity=0.25] (4,0) circle (0.5);
\end{tikzpicture}

```

Sidenote on opacity and drawing precedence: Tikz draws objects on top of each other, in the order they are given. So, if you want something to appear behind something else, the obscured object must be drawn first, i.e. above the line where the obscuring object is drawn.

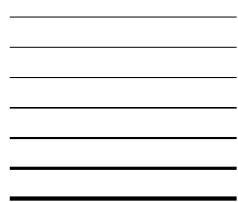
As well as passing options to individual `\draw` commands, you can pass options to the environment to set that option globally for everything in that environment. For instance, the `line width` option for all the draw commands is set to 2pt at the top, so we don't need to set it for each draw command individually.

Stroke widths affect the width of the paths drawn, and can be set with. There are various built-in widths available:

```

\begin{tikzpicture}[yscale=0.4]
  \draw[ultra thin] (0,6) -- (3,6);
  \draw[very thin] (0,5) -- (3,5);
  \draw[thin] (0,4) -- (3,4);
  \draw[semithick] (0,3) -- (3,3);
  \draw[thick] (0,2) -- (3,2);
  \draw[very thick] (0,1) -- (3,1);
  \draw[ultra thick] (0,0) -- (3,0);
\end{tikzpicture}

```



Arrows can also be added using this syntax:

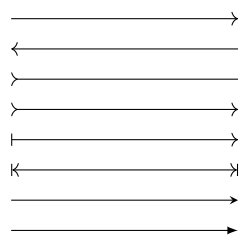
```
\draw[->] ...
```

or similar. There are also various types of arrowheads, > being one of the basic kinds. More configurations are listed below:

```

\begin{tikzpicture}[yscale=0.4]
  \draw[->] (0,7) -- (3,7);
  \draw[<-] (0,6) -- (3,6);
  \draw[>-] (0,5) -- (3,5);
  \draw[>->] (0,4) -- (3,4);
  \draw[|>] (0,3) -- (3,3);
  \draw[|<->|] (0,2) -- (3,2);
  \draw[-stealth] (0,1) -- (3,1);
  \draw[-latex] (0,0) -- (3,0);
\end{tikzpicture}

```

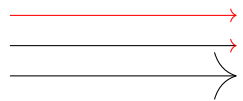


Other usual options may also be passed locally to the arrowhead:

```

\begin{tikzpicture}[yscale=0.4]
  \draw[red,->] (0,2) -- (3,2);
  \draw[-{>[red]}] (0,1) -- (3,1);
  \draw[-{>[length=3mm]}] (0,0) -- (3,0);
\end{tikzpicture}

```



Lines can also be dashed/dotted. There are various built-in pattern available (not all listed):

```

\begin{tikzpicture} [yscale=0.4]
..... \draw[dotted] (0,6) -- (3,6);
..... \draw[densely dotted] (0,5) -- (3,5);
..... \draw[loosely dotted] (0,4) -- (3,4);
----- \draw[dashed] (0,3) -- (3,3);
----- \draw[densely dashed] (0,2) -- (3,2);
- - - - - \draw[loosely dashed] (0,1) -- (3,1);
----- \draw[dash dot] (0,0) -- (3,0);
\end{tikzpicture}

```

More generally, you can specify the pattern manually using the `dash pattern` option:

`dash pattern=x on y off`

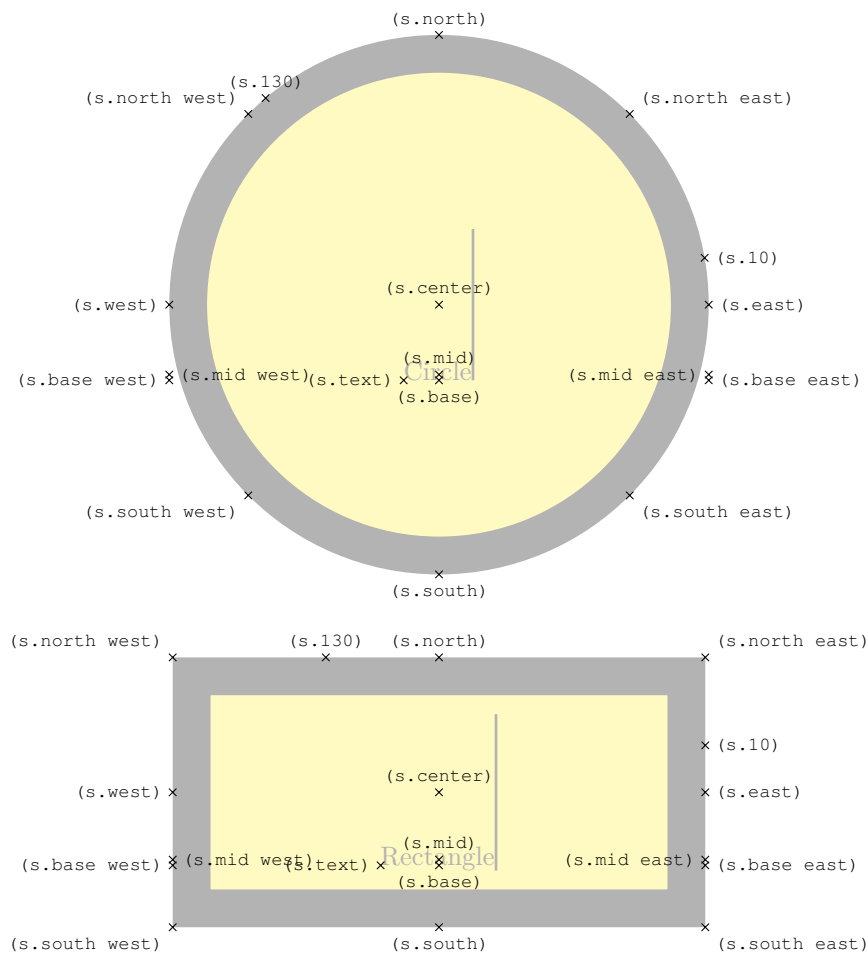
where `x` and `y` are any \TeX dimensions.

Nodes

The Calc Library

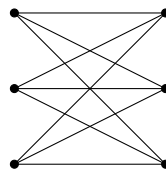
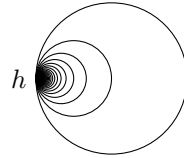
Your First Tikz Diagram

Anchors

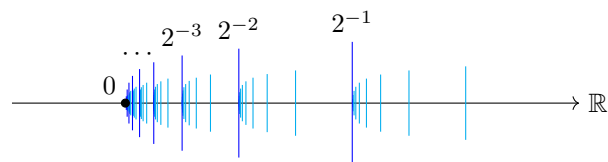


Samples

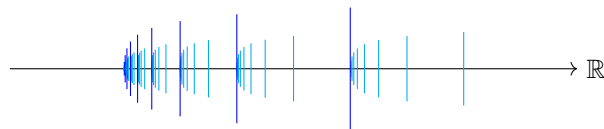
Iteration



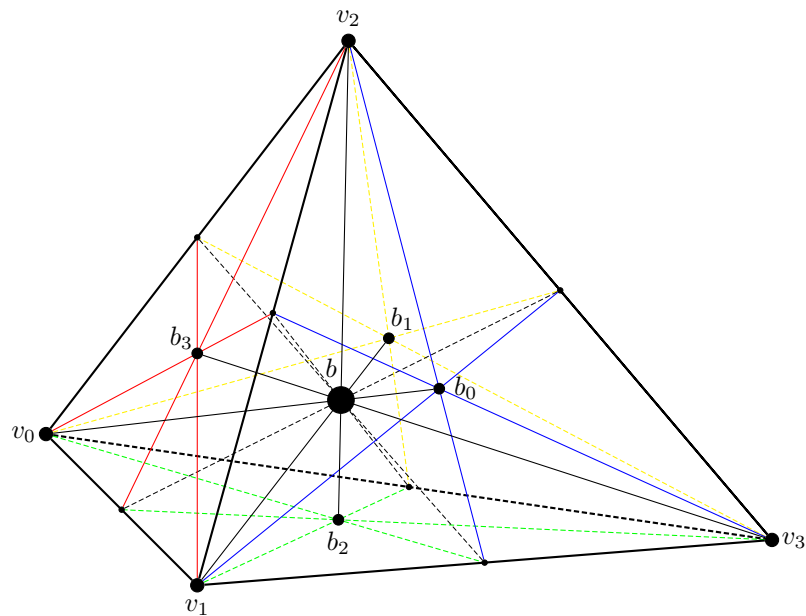
$$Z = \{0\} \cup \{2^{-n} : n \in \mathbb{Z}^+\} \cup \{2^{-n} + 2^{-n-m} : n, m \in \mathbb{Z}^+\}$$



Simpler version without if/else clauses for labels:



Calc Library



Exercises

Time to put everything together. Recreate the following diagram:

