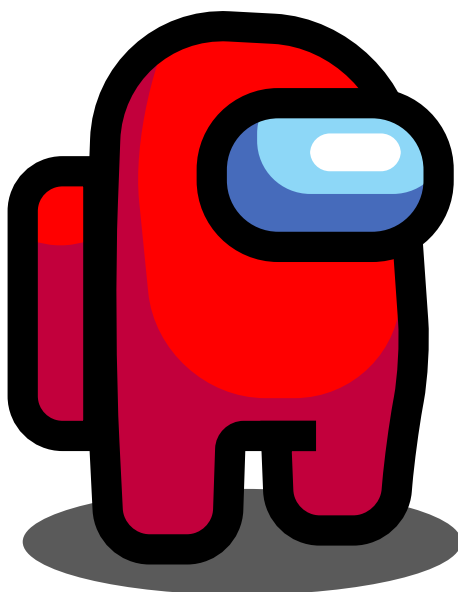# Introduction to PGF/Ti*k*z

Kit Liu

December 4, 2024

```
\begin{tikzpicture}
    \filldraw[draw=none, fill=black, opacity=0.65]
        (1.9,0.1) ellipse (2.9 and 0.7);
    \path[save path=\back]
        {[rounded corners=15pt]
        (0,5)
    -- (-1,5)
    -- (-1,1.5)}
    -- (0,1.5)
    -- cycle;
    \filldraw[red, use path=\back];
    \begin{scope}
        \clip[use path=\back];
        \filldraw[draw=none, fill=purple, opacity=0.95]
            {[rounded corners=5pt]
            (-1,4.1)
        -- (-0.5,4)}
        -- (0,4.1)
        -- (0,1.5)
        -- (-1,1.5)
        -- cycle;
    \end{scope}
    \draw[line width=4mm, use path=\back, draw=black];
    \path[save path=\body]
        {[rounded corners=15pt]
        (0,3)
    -- (0.15,0)
    -- (1.7,0)}
        {[rounded corners=5pt]
    -- (1.75,1.5)
    -- (3,1.5)}
    -- (2.35,1.5)
        {[rounded corners=15pt]
    -- (2.4,0.25)
    -- (3.9,0.25)}
        {[rounded corners=15pt]
    -- (4,1.4)
    -- (4.2,2.5)}
        {[rounded corners=45pt]
    -- (3.9,6.8)
    -- (0.15,7)
    -- cycle};
    \filldraw[red, use path=\body];
    \begin{scope}
        \clip[use path=\body];
        \filldraw[draw=none, fill=purple, opacity=0.95]
            (0,9)
        -- (1,7)
            {[rounded corners=40pt]
        -- (0.4,6)
        -- (0.8,2)
        -- (4,2)}
        -- (4,4.5)
        -- (4,0)
        -- (0,0)
        -- cycle;
    \end{scope}
    \draw[line width=4mm, use path=\body, draw=black];
    \draw[rounded corners=25pt, fill=cyan!40!white, draw=black]
        (4.5,6) rectangle (1.5,4);
    \begin{scope}
        \clip (4.5,6) [rounded corners=25pt] rectangle (1.5,4);
        \filldraw[draw=none, fill=blue!50!black, opacity=0.5]
            {[rounded corners=20pt]
            (2.1,6.2)
        -- (2.1,4.7)
        -- (4.6,4.7)}
        -- (4.6,5.2)
        -- (4.6,4)
        -- (1.5,4)
        -- (1.5,6)
            {[rounded corners=20pt]
        -- (2.5,6)
        -- cycle};
        \draw[very thick, rounded corners=7.5pt, fill=white!90, draw=none]
            (4,5.5) rectangle (2.8,5);
    \end{scope}
    \draw[rounded corners=25pt, line width=4mm, draw=black]
        (4.5,5.9) rectangle (1.5,4);
\end{tikzpicture}
```

What is PGF/Ti*k*z?

PGF/Ti*k*z 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 Ti*k*z diagram is built within the `tikzpicture` environment. Primitives are specified and drawn using Ti*k*z commands within the environment, each terminated with a semicolon (`;`).

Specifying Points

There are many ways to specify *points* or *coordinates* in Ti*k*z. Coordinates are always written with round brackets.

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.

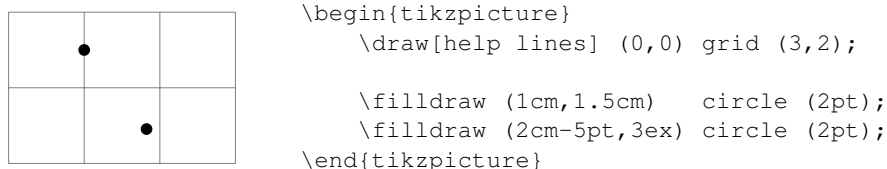For instance, cartesian coordinate may be specified explicitly using the `canvas` coordinate system:

```
(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, with the `calc` library loaded, 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}
```
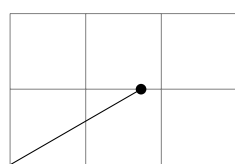
Polar coordinates may be specified using the `canvas polar` system

```
(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 I'll only expand more on 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 `(-3.85mm,-3.85mm)`.

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)].

A coordinate may also be given a name using the syntax:

```
\coordinate (name) at (coordinate)
```

This is useful for repeatedly referencing a coordinate later, or for performing computations.

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 trianglar 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

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

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

---

*It is possible to use coordinates like `(1,2cm)`, which is neither an xy-coordinate nor a `canvas` coordinate. Roughly speaking, if you have mixed coordinates, Ti*kz* will typecast the coordinate `(X,Ycm)` to the sum `(X,0)+(0pt,Y)`, and similarly for any mixed dimensions. In particular, `(2+3cm,0)` is *not* the same as `(2cm+3cm,0)`, as it will be typecast to the sum `(2pt,0) + (3cm,0) = (2pt+3cm,0)`.

specifies a rectangular box with opposite corners at `(a,b)` and `(c,d)`.
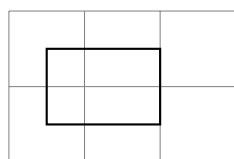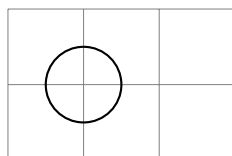
```
\begin{tikzpicture}[line width=5pt]
    \draw[help lines] (0,0) grid (3,2);

    \draw[thick] (0.5,0.5) rectangle (2,1.5);
\end{tikzpicture}
```

`(a,b) circle (r)`

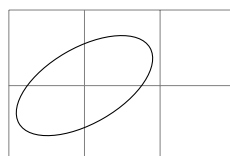specifies a circle centred at `(a,b)` with radius `r`.

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

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

`(a,b) ellipse [x radius=X, y radius=Y, rotate=T]`

specifies an ellipse centred at `(a,b)` with radii `X` and `Y`, and rotation `T`.

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

    \path[draw] (1,1) ellipse [x radius=1, y radius=0.5, rotate=30];
\end{tikzpicture}
```

Arcs

The *arc* operation draws arcs of ellipses between paths, specified using the following syntax:

```
\path ... arc [radius=R, start angle=A, end angle=B] ... ;
```

(The x and y radii may also be set separately.)

There also exists a shorter syntax for circular arcs:

```
\path ... arc (start:end:radius) ... ;
```

However, this form is very difficult to read, so the normal syntax is recommended in general.
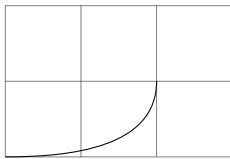
```
\begin{tikzpicture}[line width=5pt]
    \draw[help lines] (0,0) grid (3,2);

    \path[draw] (1,0) arc (0:45:1);
\end{tikzpicture}
```

To

The *to* operation adds a flexible path between coordinates that may be modified through various options. For instance, there are the `out` and `in` keys, that change the outgoing and incoming angles the path makes to the coordinates:

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

    \path[draw] (0,0) to[out=0,in=-90] (2,1);
\end{tikzpicture}
```
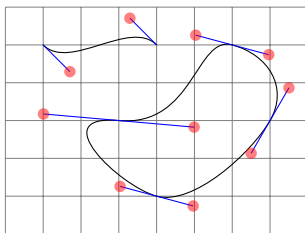
Bézier Curves

Cubic Bézier curves may be specified with the following syntax:

```
\draw (x,y) .. controls (a,b) and (c,d) .. (x,y');
```

where the coordinates `(a,b)` and `(c,d)` are the two control points of the curve.

```
\begin{tikzpicture}
    \draw [help lines] (-4, -1) grid (4, 5);

    \draw [show curve controls]
        (-3, 4) .. controls ++(135:-1) and ++(135:1) .. (0, 4);
    \draw [show curve controls] (0, 0)
    .. controls ++(165:-1) and ++(240: 1) .. ( 3, 2)
    .. controls ++(240:-1) and ++(165:-1) .. ( 2, 4)
    .. controls ++(165: 1) and ++(175:-2) .. (-1, 2)
    .. controls ++(175: 2) and ++(165: 1) .. ( 0, 0);
\end{tikzpicture}
```

It is often helpful to specify control points using relative and polar coordinates, as above.

Actions

Note that, so far, a path is simply a series of curves and coordinates – we haven't actually told Ti*kz* 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.
```

Without one of these options, a path will simply be discarded:

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

Fills for self-intersecting paths is somewhat complicated; I'll only discuss this on request.

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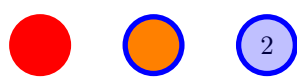
- colours;

- 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).

Colours may be specified using the xcolors syntax (e.g. blue!50!white). (This will be more important when we introduce iteration.)

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

```
\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:
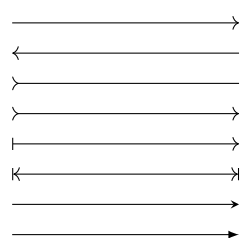
```
\begin{tikzpicture}[yscale=0.4]
    \draw[ultra thin]  (0,6) -- (3,6);
    \draw[very thin]   (0,5) -- (3,5);
    \draw[thin]        (0,4) -- (3,4);
    \draw[semithick]   (0,3) -- (3,3);
    \draw[thick]       (0,2) -- (3,2);
    \draw[very thick]  (0,1) -- (3,1);
    \draw[ultra thick] (0,0) -- (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 (list is not exhaustive):
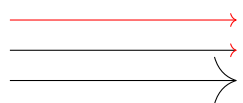
```
\begin{tikzpicture}[yscale=0.4]
    \draw[->]          (0,7) -- (3,7);
    \draw[<-]          (0,6) -- (3,6);
    \draw[>-]          (0,5) -- (3,5);
    \draw[>->]         (0,4) -- (3,4);
    \draw[|->]         (0,3) -- (3,3);
    \draw[|<->|]       (0,2) -- (3,2);
    \draw[-stealth]    (0,1) -- (3,1);
    \draw[-latex]      (0,0) -- (3,0);
\end{tikzpicture}
```

Other usual options may also be passed locally to the arrowhead:
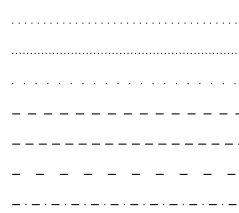
```
\begin{tikzpicture}[yscale=0.4]
    \draw[red,->]             (0,2) -- (3,2);
    \draw[-{>[red]}]          (0,1) -- (3,1);
    \draw[-{>[length=3mm]}]   (0,0) -- (3,0);
\end{tikzpicture}
```

Lines can also be dashed/dotted. There are various built-in patterns available (list is not exhaustive):

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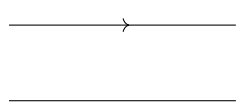
More generally, you can specify the pattern manually using the `dash pattern` option:

```
dash pattern=x on y off
```

where `x` and `y` are any TEX dimensions.

Lines may also be *decorated* as follows:

```
\begin{tikzpicture}[decoration={
    markings,
    mark=at position 0.53 with {\arrow{>}}}
    ]
    \draw[postaction={decorate}] (0,1) -- (3,1);
    \draw                        (0,0) -- (3,0);
\end{tikzpicture}
```
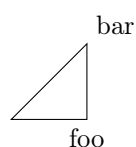
Note that arrow decorations are centred at their tips, so arrows need to be positioned slightly past the halfway point (e.g. 0.53 in the above) to appear properly centred.

Nodes

Nodes are used to add text to a diagram. They may be specified and drawn using the following syntax:

```
\node at (x,y) [options] {label}
```

They may also be added to paths using the special path operation `node`:
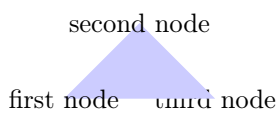
```
\begin{tikzpicture}
    \draw
        (0,0)
        -- (1,0) node [below]       {foo}
        -- (1,1) node [above right] {bar}
        -- cycle;
\end{tikzpicture}
```

Nodes are specified in this way are positioned at the last coordinate mentioned. Note that nodes are specified in this way are also *not* part of the path itself; they are separate elements drawn before or after the path:
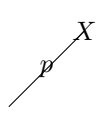
```
\begin{tikzpicture}[fill=blue!20!white]
\fill
    (0,0) node                  {first node}
    -- (1,1) node               {second node}
    -- (2,0) node [behind path] {third node};
\end{tikzpicture}
```

Nodes may also be placed on paths by specifying the node after a path starts, but before the endpoint is specified:
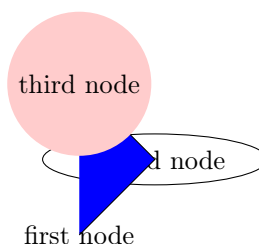
```
\begin{tikzpicture}
\begin{tikzpicture}
\draw
    (0,0) -- node {$p$}
    (1,1) node {$X$};
\end{tikzpicture}
```

Nodes may also take some standard shapes as options (as well as colours, opacities, etc.). Passing the `draw` option along with these shapes will draw the shape around the node, dynamically resizing to fit the label inside:

```
\begin{tikzpicture}
    \draw[fill=blue]
        (0,0) node                              {first node}
     -- (1,1) node[ellipse,draw, behind path] {second node}
     -- (0,2) node[circle,fill=red!20]        {third node};
    \end{tikzpicture}
```

Nodes may also be given names:

```
\node (name) at (x,y) {label}
\path ... -- (x,y) node (name) {label} -- ...
```
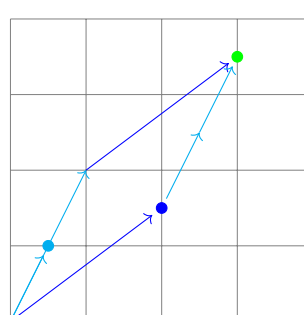
A named node may be later referenced by its name. This is useful for computation on coordinates, etc.

Coordinate Computation

We have already seen some computation within coordinates, but we can do a lot more than simply adding dimensions together.

A coordinate computation is marked with a pair of $ symbols. One basic computation is to add and scale coordinates:

```
\begin{tikzpicture}
    \draw[help lines] (0,0) grid (4,4);

    \filldraw[blue] (2,1.5) circle (2pt) node (a) {};
    \filldraw[cyan] (0.5,1) circle (2pt) node (b) {};
    \filldraw[green] ($(a) + 2*(b)$) circle (2pt);

    \draw[blue,->] (0,0) -- (a);
    \draw[blue,->] ($2*(b)$) -- ($(a) + 2*(b)$);

    \draw[cyan,->] (0,0) -- (b);
    \draw[cyan,->] (0,0) -- ($2*(b)$);
    \draw[cyan,->] (a) -- ($(a) + (b)$);
    \draw[cyan,->] ($(a) + (b)$) -- ($(a) + 2*(b)$);
\end{tikzpicture}
```

A coordinate may be followed by various kinds of modifiers. The first kind of interest is the *partway modifier*, with syntax:

```
(a)!N!(b)
```

The number `N` linearly interpolates between the first and second coordinates:

$$(1 - N)(a) + N(b)$$

Note that `N` need not be between 0 and 1.

If `N` instead has a dimension, e.g. `N=1cm`, then this is the *distance modifier*, and instead returns the point at distance `N` from `(a)` along the line connecting `(a)` and `(b)`.

If `N` is instead itself is a coordinate, e.g. `N=(c)`, then this is the *projection modifier*, and instead returns the projection of `N` on to the line connecting `(a)` and `(b)`.
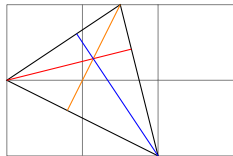
```
\begin{tikzpicture}
    \draw [help lines] (0,0) grid (3,3);
    \coordinate (a) at (1,0);
    \coordinate (b) at (3,2);

    \draw[->] (a) -- (b);
    \coordinate (c) at ($ (a)!1! 10:(b) $);
    \draw[->,red] (a) -- (c);
    \fill ($ (a)!.5! 10:(b) $) circle (2pt);
\end{tikzpicture}
```

Another kind of modifier is the *angle modifier*, with syntax:

```
(a)!N!T:(b)
```

(`N` cannot be omitted from this; just set `N=1` if the other modifiers are not desired.)

The number `T` rotates the line from `(a)` to `(b)` about the point `(a)` by `T` degrees:
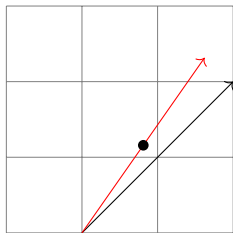
```
\begin{tikzpicture}
    \draw [help lines] (0,0) grid (3,3);
    \coordinate (a) at (1,0);
    \coordinate (b) at (3,2);

    \draw[->] (a) -- (b);
    \coordinate (c) at ($ (a)!1! 10:(b) $);
    \draw[->,red] (a) -- (c);
    \fill ($ (a)!.5! 10:(b) $) circle (2pt);
\end{tikzpicture}
```
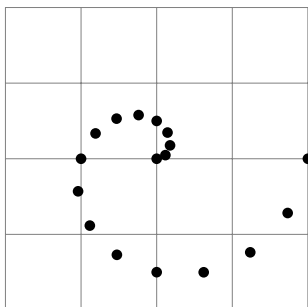
```
\begin{tikzpicture}
    \draw [help lines] (0,0) grid (4,4);

    \foreach \i in {0,0.125,...,2} {
        \fill ($(2,2) !\i! \i*180:(3,2)$) circle (2pt);
    }
\end{tikzpicture}
```

Iteration
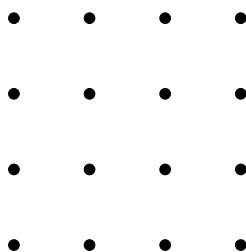
Plots

For small plots, you can use the special path operation `plot`. This requires you to handle a lot of things manually, like adding axes/ticks, but is suitable for simplistic plots.

The `plot` command can be used to connect a list of coordinates together:

```
\begin{tikzpicture}
    \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
\end{tikzpicture}
```

This isn't very useful by itself, since we could have just specified a path and stroked it manually, but this command can also read coordinates from external files.

The `plot` command can be used to plot a function parametrically by giving a coordinate in terms of the macro `\x`:
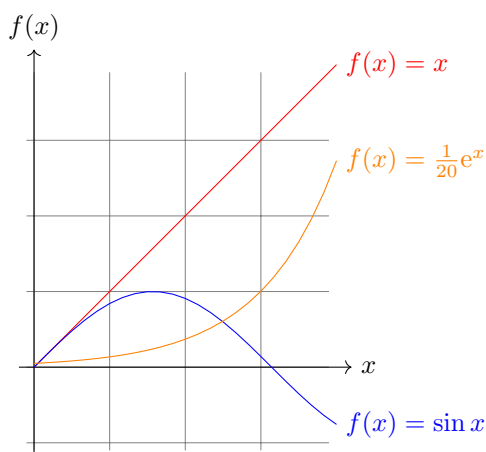
```
\begin{tikzpicture}[domain=0:4]
    \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

    \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
    \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

    \draw[color=red]
        plot (\x,\x)                node[right] {$f(x)=x$};
    \draw[color=blue]
        plot (\x,{sin(deg(\x))}) node[right] {$f(x) = \sin x$};
    \draw[color=orange]
        plot (\x,{0.05*exp(\x)}) node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```
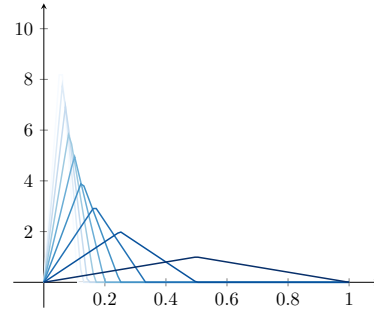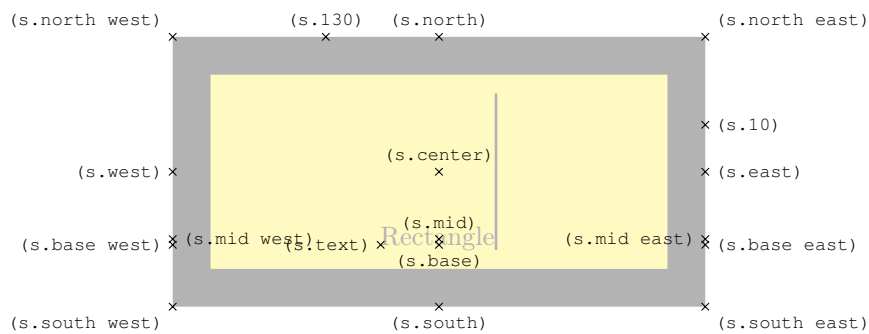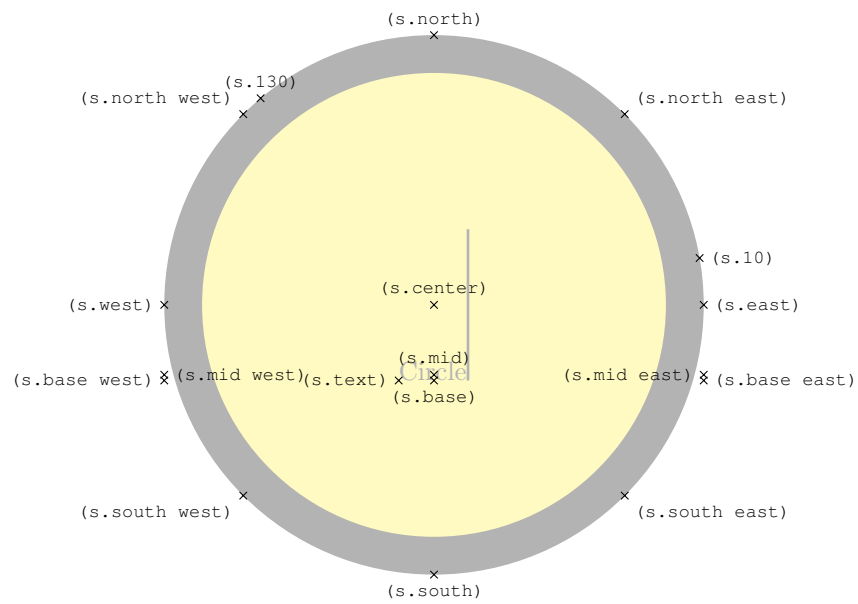
Axis Environment

ColorBrewer

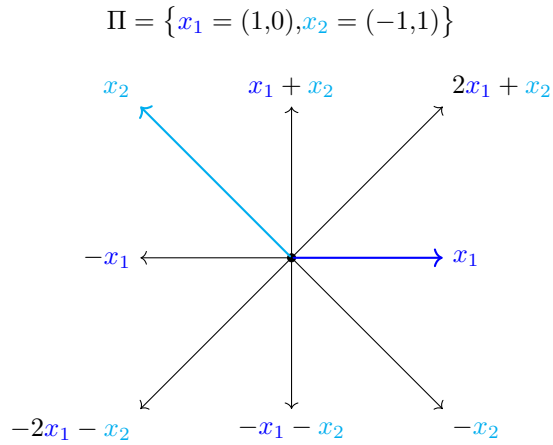$$g_n(x) = \begin{cases} 2nx & x \in [0, \frac{1}{2n}) \\ -2n\big(x - \frac{1}{n}\big) & x \in [\frac{1}{2n}, \frac{1}{n}) \\ 0 & x \in [\frac{1}{n}, 1] \end{cases} \qquad h_n(x) = \begin{cases} 2n^2x & x \in [0, \frac{1}{2n}) \\ -2n^2\big(x - \frac{1}{n}\big) & x \in [\frac{1}{2n}, \frac{1}{n}) \\ 0 & x \in [\frac{1}{n}, 1] \end{cases}$$
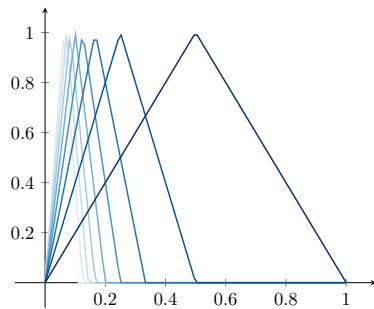
Anchors

(s.north)

(s.130)

(s.north west)   (s.north east)

(s.10)

(s.center)

(s.west)   (s.east)

(s.mid)

(s.base west) (s.mid west)(s.text) Circle (s.mid east) (s.base east)

(s.base)

(s.south west)   (s.south east)

(s.south)

(s.north west)   (s.130)   (s.north)   (s.north east)

(s.10)

(s.center)

(s.west)   (s.east)

(s.mid)

(s.base west) (s.mid west)(s.text) Rectangle (s.mid east) (s.base east)

(s.base)

(s.south west)   (s.south)   (s.south east)

Exercises

1. Recreate the following diagram:

$$\Pi = \big\{ x_1 = (1,0), x_2 = (-1,1) \big\}$$

2. Recreate the following diagrams:
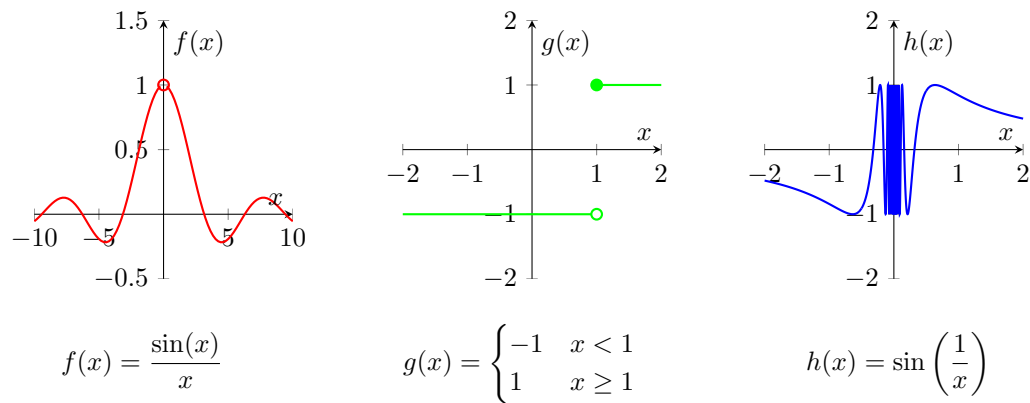

$$\Delta^0 \qquad\qquad \Delta^1 \qquad\qquad \Delta^2$$
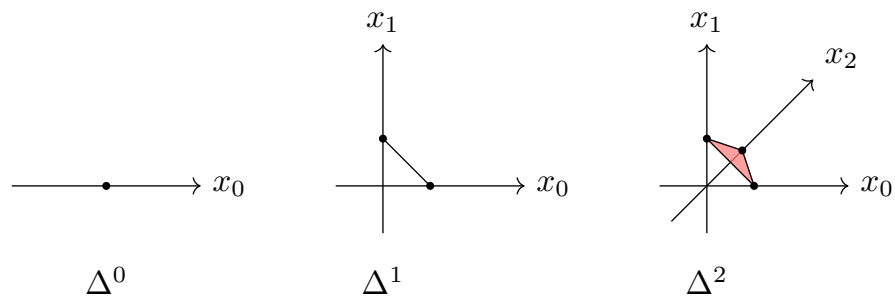
3. Recreate the following diagram:
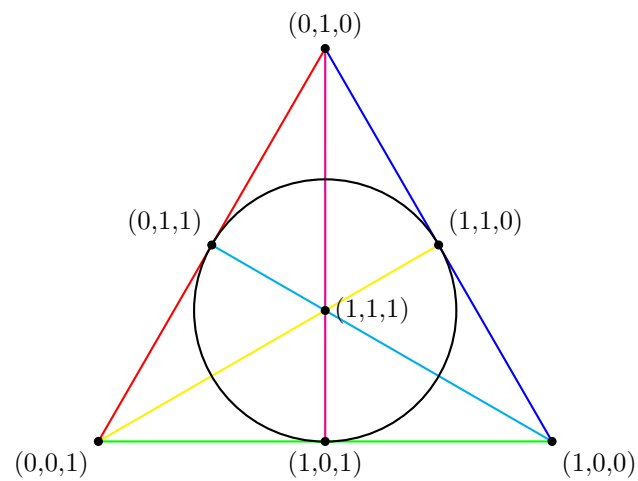
4. Recreate (some of) the following diagrams:
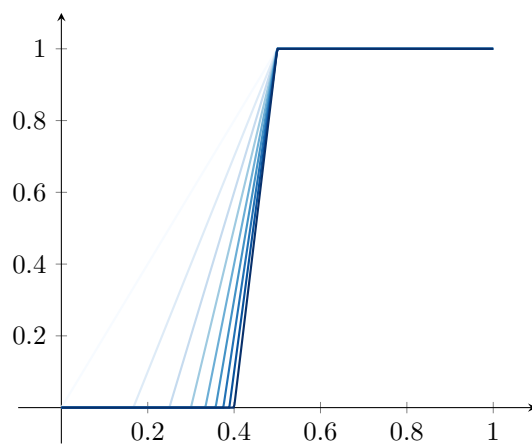
5. Recreate (some of) the following diagrams:

$$f(x) = \frac{\sin(x)}{x} \qquad g(x) = \begin{cases} -1 & x < 1 \\ 1 & x \geq 1 \end{cases} \qquad h(x) = \sin\left(\frac{1}{x}\right)$$
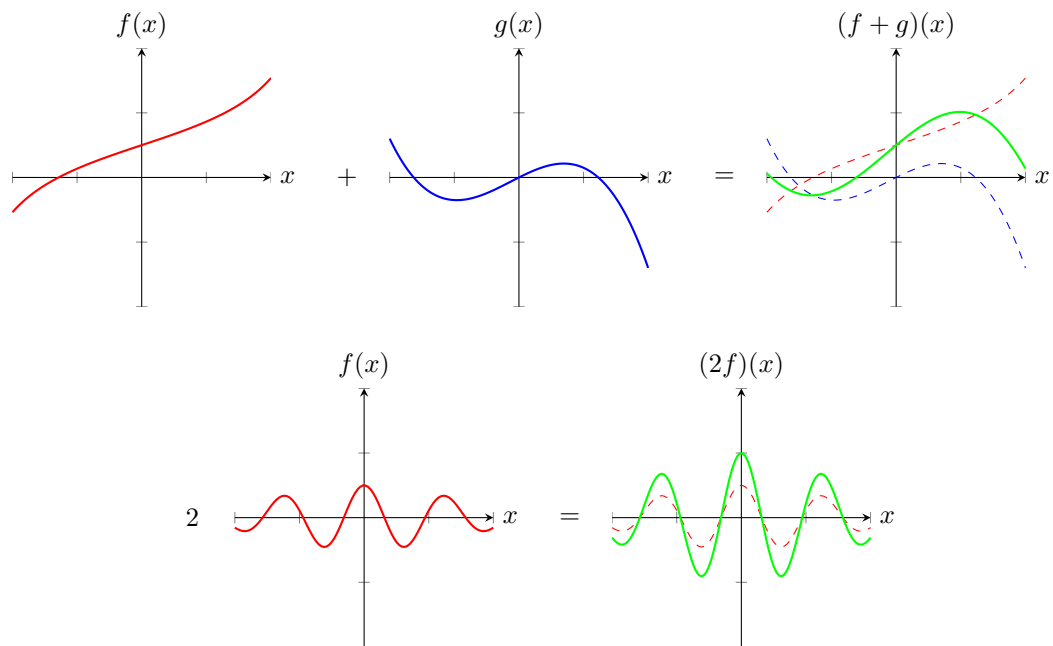
6. Recreate the following diagram:

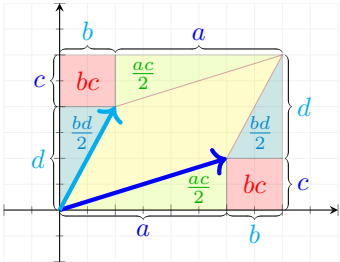$$\mathbb{RP}^2 \qquad\qquad v = a + b \qquad\qquad \mathbb{RP}^2 \subseteq U \cup V \qquad\qquad U \cap V$$
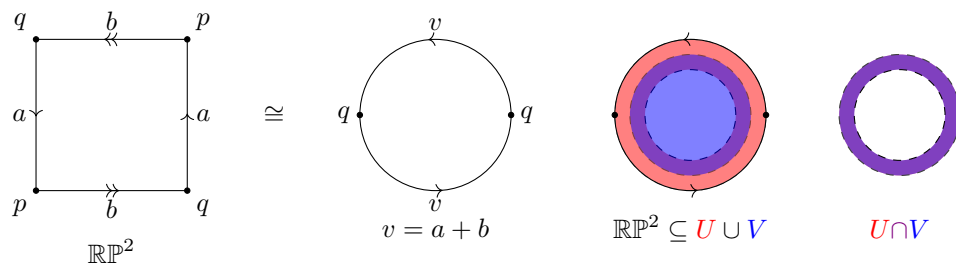
7. Recreate the following diagram:

$$f_n(x) = \begin{cases} 0 & 0 \leq x < \frac{1}{2} - \frac{1}{n} \\ 1 - n(\frac{1}{2} - x) & \frac{1}{2} - \frac{1}{n} \leq x < \frac{1}{2} \\ 1 & \frac{1}{2} < x \leq 1 \end{cases}$$
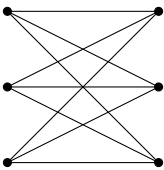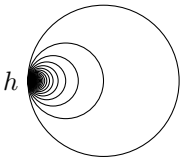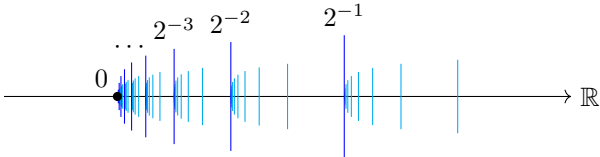
Samples

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = (a + b)(c + d) - 2\left(\frac{ac}{2}\right) - 2\left(\frac{bd}{2}\right) - 2bc = ad - bc$$
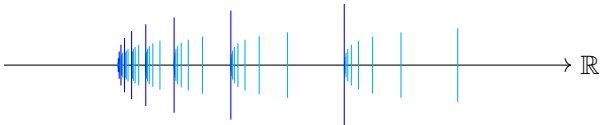
Iteration

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

Simpler version without if/else clauses for labels:

Calc Library