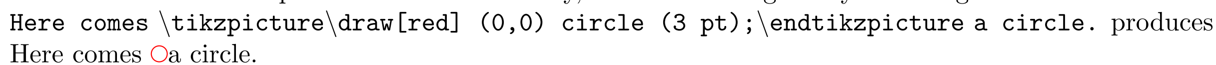
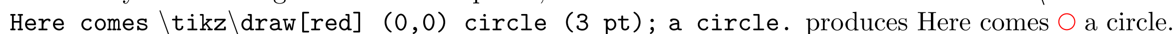
PGF a TikZ

PGF is a package of macros and processing programs for drawing pictures, while TikZ provides a user-friendly frontend. It is loaded using `\input tikz.tex` in plain TeX/$\mathcal{A}_\mathcal{M}\mathcal{S}$-TEX, LateX users have their own way. This loads the engine and also some basic libraries. For extended functionality one can load extra libraries. This is done using the control word `\usetikzlibraries`, where we specify libraries in its argument as a comma-separated list. Example: `\usetikzlibraries{arrows,mathematics}`. Just one can be loaded using `\usetikzlibrary`.
With a bit of luck this works and we can move to the next chapter. However, sometimes the default setup does not work as needed. The basic dvi format does not actually have any graphics capability and just passes graphics instructions to the follow-up driver. The problem is that there is no accepted standard for this; moreover, drivers differ in their graphics capabilities, and thus the outcome depends on how well those graphical instructions match the driver's expectations. The default setting seems to work well when running `tex` or `latex` and then `dvips`, it also works with `pdftex`. If troubles appear, we can try to select a suitable driver by placing the following code **before** the `\input tikz` line:
`\def\pgfsysdriver{pgfsys-pdftex.def}` offers the most complete functionality, intended for `pdftex`.
`\def\pgfsysdriver{pgfsys-dvips.def}` is for use with `tex` followed by `dvips`. Offers most of PGF features except image inclusion and shading is of lower quality.
`\def\pgfsysdriver{pgfsys-tex4ht.def}` is for use with `tex4ht`. Restricted set of features: no image inclusion or remembering of pictures, plain text only, no matrices, no shading etc.
`\def\pgfsysdriver{pgfsys-dvi.def}` is for producing perfectly standard `dvi` files, which means that the only tool for graphics are small black rectangles. Accordingly it can only do lines and plain curves, and that by approximating them with tiny rectangles. No filling, shading or anything. Of course, placing text is possible, also transformations are supported.

1. Structure, coordinates

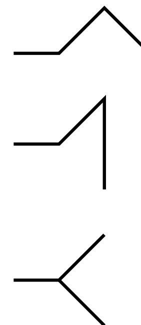
Individual pictures are created in blocks delimited by `\tikzpicture` and `\endtikzpicture`, in LateX they use `\begin{tikzpicture}` and `\end{tikzpicture}`. A typical `tikzpicture` consists of commands, each must be ended with a semicolon `;` and every command including \tikzpicture can include optional parameters enclosed in square brackets `[ ]`. It is not possible to use drawing commands outside the `tikzpicture` environment.
Each `tikzpicture` block creates a picture that is treated by TEX as a standard horizontal box, so it can be a part of a line and we can manipulate it in the usual way, for instance align it by centering.

`Here comes \tikzpicture\draw[red] (0,0) circle (3 pt);\endtikzpicture a circle.` produces
Here comes ○a circle.
When only one drawing command is required, we can use the shortened form that uses `\tikz`.

`Here comes \tikz\draw[red] (0,0) circle (3 pt); a circle.` produces Here comes ○ a circle.
The resulting box has TEX dimensions determined by points used in the construction, even auxiliary points that are not drawn, so sometimes the box is too large. This can be fixed, see `use as bounding box` in chapter 3. Drawing. The baseline position of this box is its lower edge by default. This can be also changed, see parameters of `\tikzpicture`.

Coordinates

The `\tikzpicture` command creates an empty canvas with an origin and several coordinate systems. Coordinates of points are supplied as vectors in rounded parentheses. The full format is `(`*coordinate system* `cs:`*data*`)`, but typically we prefer implicit forms.
A pair `(`x`,`y`)` is interpreted as a two-dimensional Carthesian coordinate, with x pointing right and y pointing up. This is called the `canvas` coordinate system. Numbers without units are interpreted as centimeters, but this can be changed, see Parameters below. Traditional units like `cm`, `mm`, `pt` are accepted. The engine evaluates dimensions, so one can supply also expressions like `2+3`. Warning! When we mix data with and without units, then those without are interpreted as points! For instance, the coordinate `(2+3cm,15)` is interpreted as (2pt+3cm,15pt). However, `(2+3,15)` is interpreted as (2cm+3cm,15cm)
A pair `(`a`:`r`)` is interpreted as a polar coordinate with angle a and radius r. The angle is measured in degrees. The radius can include a unit. Degree zero points in the direction of the x-axis, the y-axis is 90 degrees. Some angles have word equivalents, namely `up`, `down`, `right`, `left`, `north`, `south`, `east`, `west`, `north east`, `north west`, `south east`, `south west`. The radius component can have two components, say, `(30:1cm and 2 cm)`. This sets up an elliptic coordinate system, here we are specifying a point that is the intersection of a ray at 30 degrees with the ellipse with the x-half-axis 1 cm and y-half-axis 2 cm.
A triple `(`x`,`y`,`z`)` is interpreted as a three-dimensional Cartesian coordinate drawn in the standard perspective. The x-coordinate points right, the y-coordinate points up, and the z-coordinate is printed at the position $\bigl(-\frac{1}{\sqrt{2}}z, -\frac{1}{\sqrt{2}}z\bigr)$.
Different coordinate systems can be combined freely within one drawn object. We will use *point* as a reference to a coordinate of any type.
A coordinate can include options after the opening parenthesis, e.g. `([xshift=1pt,rotate=45]3,7)` that apply to this particular point.
There is a special coordinate system called `node`. We can assign names to locations and then refer to them. Moreover, we can also assign a name to some types of objects and then refer to locations on their borders. For more details, see chapter 5. Referring to locations.
There are more coordinate systems with many customizations possible, see the official manual.

When drawing paths, TikZ keeps track of the latest **current location**. In fact, there are two current locations, one for the purpose of drawing and the other for the purpose of referring to. Typically, these two agree, and they are updated after every drawing operation.

When a coordinate is preceded by a plus sign, it becomes a **relative reference** with respect to the latest reference location. For instance, while `(1,2)` refers to a location with respect to the origin, writing `+(1,2)` in a situation when we previously placed a virtual pen at (3,-1) refers to the coordinate (4,1). There are actually two versions. The version `++(,)` updates both the curent drawing location and the current reference location, so it naturally fits with typical drawing operations. On the other hand, the version `+(,)` sets up a new drawing location, but does not update the current reference location.



In the following two examples, the reference point established after drawing the first segment is $(1,1)$. The first command (the top picture) produces a naturally chained curve. In the second command, the third segment started at the natural place (the latest drawing point was updated), but it ends at a different place as the latest reference point was not upgraded when drawing the second segment.

Things can be further complicated by the fact that some operations do not update locations at all, in particular the `edge` operation used in the third code. Note that it even overrides the location changing form `++`.

```
\draw[very thick] (0,0) -- (1,0) -- ++(1,1) -- ++(1,-1);
\draw[very thick] (0,0) -- (1,0) -- +(1,1) -- ++(1,-1);
\draw[very thick] (0,0) -- (1,0) edge ++(1,1) -- ++(1,-1);
```

Relative coordinates also get a special treatment when drawing Bezier curves.

2. Parameters

The behaviour of TikZ is controlled by parameters. They are set using `parameter=`*value* at four possible levels. The control word `\tikzset` that can be used anywhere in our `tex` file (even outside the `tikzpicture` environment) changes the parameters for all `tikzpicture` blocks that follow. Second, we can change the settings for a specific `\tikzpicture` by specifying values in the optional argument of this command. Third, some commands in one `tikzpicture` block may be grouped using `\scope;` and `\endscope;` and options in `\scope` set parameters for this group. Finally, we can change the settings individually for each command in its options. The order of this list also indicates a natural priority. More options are separated by commas.

A typical example:

```
\tikzpicture[shorten >=2pt,color=black,scale=0.7]
  \draw[->,thick,double,color=red](0,0) -- (1,1);
\endtikzpicture
```

This will draw an arrow going up and right, using doubled line, and also using thick lines and red colour. The arrow will actually stop short of the two specified points by 2 pts. The whole picture will be scaled down to 70 %. Note that `thick` is a style that defines a parameter (line width), while `color=` is a direct parameter definition.

When a group of parameter settings is used often, we can make them into a named **style** and call it by its name. Style definitions are done as options for `\tikzpicture`.

```
\tikzpicture[myarrow/.style={->,thick,color=red}]
\draw[myarrow] (0,0) -- (1,1);
```

There is a special style that allows for global setting of parameters. If we, say, set `\tikzset{line width=1pt}`, then there is a danger that this can be changed later. The style `every picture` is automatically installed at the beginning of every picture, so it is better to put `\tikzset{every picture/.style={line width=1pt}}`.

Note that a style may also feature a parameter. After `\tikzset{mycol/.style={draw=black,fill=#1}}` we can use `\draw[mycol=red] (0,0) rectagle (2,1);`

In this chapter we will focus on parameters that can be set globally (although they are often also used locally). Parameters specific to individual operations are covered at the appropriate place.

2a. Drawing parameters

• `color=`*color* specifies the colour used for making the picture (fill, drawing, text, but not shading). Standard values for *color* are `white`, `black`, `red`, `green`, `blue`, `cyan`, `magenta`, `yellow` and `none`, a special color that can be used e.g. to interrupt a path, or supress drawing the outline of a filled picture. It is also possible to just put the name of one of these colors as an option and it is interpreted as a `color` specification.

The implementation is based on LaTeX color naming and `xcolor` extension, so LaTeX users can use also other names like `gray` and other tricks. Some of these may also work for plain users; in particular, there is a good chance that the mixing model for two colors will work. The format is *color*`!`*percentage*`!`*color*, and the first named color is mixed in at the specified percentage, with the second color supplying the rest. For instance, `\path[red!50!blue]` will draw and fill in violet. When the second color (and the exclamation mark) is not specified, then it is assumed to be white, producing lighter shades, e.g. `\path[blue!50]`.

When this fails, there is a mechanism allowing for custom colors that supports `rgb` and `gray` modules. One can define a new color using e.g. `\definecolor{myorange}{rgb}{1,0.5,0}`, or create a new color based on old one, say, `\colorlet{lightgray}{black!25}`. Note that these are TEX commands, so we do not end them with a semicolon. Then we can use the new color name as usual, say, `fill=myorange`. We can also, within a `tikzpicture` block, switch the color on using `\color{`*orange*`}`. The influence of this can be restricted by a group { }.

• `draw=`*color* specifies the colour used for drawing, `fill=`*color* specifies the colour used for filling, e.g. `\tikzpicture[draw=black,fill=blue!80!black]`. Note that these options overlap with `color`, so when, say, both `color` and `fill` are specified, then the specification that comes last applies to fills, similarly for `draw`.
• `draw opacity=`, `fill opacity=`, and `text opacity=` define opacity of these actions, and `opacity=` sets it for all. The argument ranges between 0 and 1. There are also several styles, ranging from `transparent` (`opacity=0`) through `semitransparent` or `nearly opaque` to the default `opaque`.

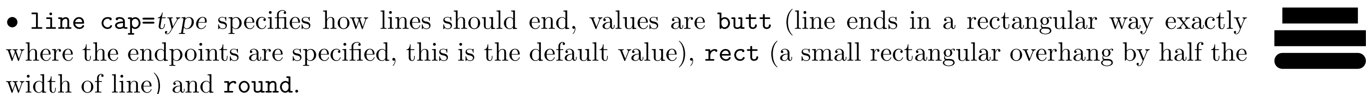
• `line width=`*dim*, where *dim* is the standard TEX dimension. The line width can be also set using one of predefined styles, we can use `ultra thin` (sets `line width=0.1pt`), `very thin`, `thin` (sets `line width=0.4pt`, this is the default value), `semithick`, `thick` (sets `line width=0.8pt`), `very thick`, or `ultra thick`.

`ultra thin` `very thin` `thin` `semithick` `thick` `very thick` `ultra thick`

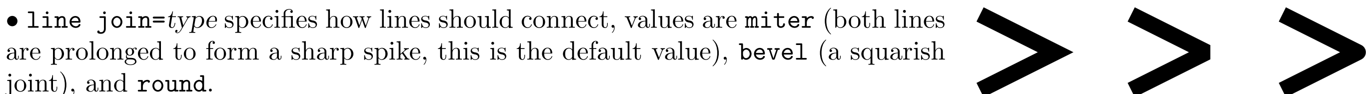
• `dash pattern=`*dashpattern* sets up the way in which lines are drawn. The pattern is set up by combining basic units of the form `on` *dim* and `off` *dim*. Example: `dash pattern=on 10 pt off 3 pt on 6 pt off 3 pt`.

There are some predefined styles that can be used instead of this specification. `solid` is the default, so `\draw` is the same as `\draw[solid]`. Other values are `dotted`, `densely dotted`, `loosely dotted`, `dashed`, `densely dashed`, `loosely dashed`.
Whatever way we use for setting up the pattern, we can always use `dash phase=`*dim*, this starts the line by going *dim* into the pattern (that is, as if the pattern was shifted to the left).

• `line cap=`*type* specifies how lines should end, values are `butt` (line ends in a rectangular way exactly where the endpoints are specified, this is the default value), `rect` (a small rectangular overhang by half the width of line) and `round`.

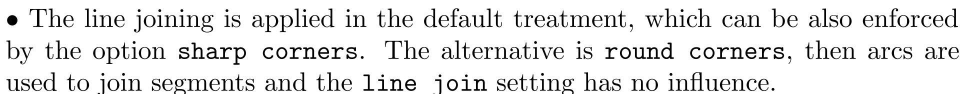


• `line join=`*type* specifies how lines should connect, values are `miter` (both lines are prolonged to form a sharp spike, this is the default value), `bevel` (a squarish joint), and `round`.



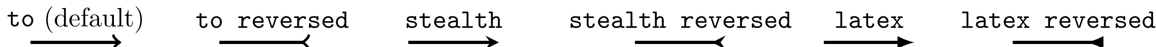
When lines meet too sharply, the protruding point in `miter` may go too far. If it goes further then `mitter limit` times width of the line, then automatically this particular joint is changed to type `bevel`. The default value is `miter limit=10`.

• The line joining is applied in the default treatment, which can be also enforced by the option `sharp corners`. The alternative is `round corners`, then arcs are used to join segments and the `line join` setting has no influence.



• `shorten >=`*dim* shortens all lines so that their ends finish a little bit (by *dim*) before the specified endpoint.
• `shorten <=`*dim* shortens beginnings of all lines.

• `>=`*arrow* and `<=`*arrow* set the meaning for arrowheads as used by `\path` (see below). For instance, if we want all arrows to be of the type `stealth`, then we set globally `>=stealth` and `<=stealth` and then we can use things like `<->>` freely (see `\draw`). The following six are the default types.



Many more types are available in the `arrows` library that also adds possibilities to modify the arrows.

2b. Transformations

The whole picture may be transformed by optional arguments of `\tikzpicture`. Most of these can be applied also to groups or individual commands, or even individual components of paths. When more transformations are specified, then they are applied fromt the last to the first in the list.
• `x=`*dim* and `y=`*dim* redefine the default unit in coordinates. For instance, `x=2cm,y=1cm`. It is also possible to specify vectors, thus setting up a general two-dimensional basis. These vectors have to be closed in groups to protect the comma from being processed as option separator: `x={(1cm,0cm)},y={(1cm,1cm)}`.
• `xshift=`*dim* shifts the picture to the right by *dim*. As usual, the default unit is `cm`.
• `yshift=`*dim* shifts the picture up by *dim*.
• `shift=`*point* shifts by the vector given by *point*, for instance `\draw[shift={(1,1)},blue]`
• `rotate=`*angle* rotates about $(0,0)$ by the *angle* given in degrees counterclockwise.
• `rotate around={`*angle*:*point*`}` rotates the whole picture by *angle* around the specified *point*.
• `scale=`*factor* scales the whole coordinate system *factor*-times, so `scale=1` leaves the picture as it is and `scale=-1` creates a picture symmetric with respect to the origin. Note that scaling affects coordinates, so it does not apply to content of nodes (see 3. Placing text) nor to width of lines.
• `xscale=`*factor* scales all x-coordinates *factor*-times. `xscale=-1` does a horizontal flip.
• `yscale=`*factor* scales all y-coordinates *factor*-times. `yscale=-1` does a vertical flip.
• `scale around={`*factor*:*point*`}` scales the whole picture *factor*-times, but with respect to the specified *point*.
• `xslant=`*factor* slants x-coordinates by the given *factor*, that is, the x-coordinate is shifted right by *factor* times y-coordinate.
• `yslant=`*factor* does the same for y-coordinates.
• `shift only` takes no value and it cancels all transformations apart from shifts.

All these are especialy useful when used with `\scope`, in this way one can move whole sections of a picture.

```
\tikzpicture
\draw...;
\scope[xshift=5cm,rotate=30];
  \draw...;
\endscope;
\endtikzpicture
```

As we define transformations, they all combine to form a coordinate transformation matrix. One can actually set it up directly,

• `cm=`$\{a,b,c,d,\mathit{point}\}$ sets up the transformation $A\vec{x}+\mathit{point}$, where $A=\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

• `reset cm` has no argument and clears the transformation matrix, this should be used with utmost care, since it destroys not just the local matrix, but also all matrices inherited from surrounding scopes.

• `baseline=` determines the position of baseline in the resulting picture box. The value `baseline=`*dim* sets the baseline to level as determined by *dim* in the y-dimension. The other form is `baseline=`*point*, where the baseline is set at the y-coordinate of the *point* as it is at the end of the drawing (after transformations etc. were applied). This point can be symbolic, e.g. `baseline=(X.base)` or `baseline=(X.north)`, where X is an object drawn and named in the `tikzpicture` group. One popular option is `baseline=current bounding box.north`.

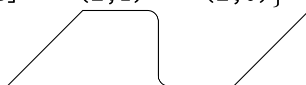
3. Drawing

The basic tool for drawing is the command `\path`. It constructs an imaginary path according to specifications (see below), and then takes action based on optional arguments. Other optional arguments can further modify the action; most of those from chapter 2 can be used, some path-specific options are described below. While the order of options is irrelevant, it is customary and advisable to put the action specification first, next to `\path`. A typical path command has the form

`\path[`*actions, options*`]` *point operation data operation data* ... `;`

Most options only apply globally to the whole path. There are several exceptions. The option `rounded corners` can be also placed within a path before some operation and influences the rest, we can also restrict its validity using a group. Similarly, we can place before some operation an optional argument featuring transformations and they will apply to all point specifications that follow.

```
\draw (0,0) -- (1,1) {[rounded corners] -- (2,1) -- (2,0)} -- (3,0) [yshift=1cm] -- (4,0);
```



Note that if we did not specify a unit in the `yshift`, then it would be interpreted as `1pt`.
Finally, we can use `edge` (see below) to apply parameters to parts of a path.

Possible **actions** are `draw`, `fill`, `clip`, `shade`, `pattern`, and `use as bounding box`. These options may accumulate, and unless you use some of these, the output of `\path` is nothing.
The actions `draw` and `fill` can be also used to set color, so we can write `\path[draw=red]`.
For frequently used actions there are shortcuts.

• `\draw` is a shortcut for `\path[draw]`, it draws the specified path.

• `\fill` is a shortcut for `\path[fill]`, it fills the inside parts of the specified path.
If the path is not closed, then it is first virtually closed. Since a path may include circles, rectangles etc, its “inside” may be a complicated thing. There are two possibilities to determine this, both are based on the winding number. Given some closed region in plane, pick an arbitrary point inside it and shoot a ray to infinity. Count how many times the curve crosses this ray, with $+1$ if the curve crosses anticlockwise, and -1 if it crosses clockwise. The resulting sum is the winding number of this region.
In the optional parameter we can specify which rule is to be used. When `\fill[nonzero rule]` is used, then any region with non-zero winding number is considered inside. When `\fill[even odd rule]` is used, then regions with odd winding numbers are considered to be inside.

• `\filldraw` is a shortcut for `\path[fill,draw]`, it first fills and then draws the specified path.

• `\pattern` is a shortcut for `\path[pattern]`, it fills the inside of the path with a pattern. The inside is determined just like in `\fill` according to the rule given as optional argument.
The pattern is specified using `pattern=`*pattern*, values are for instance `dots`, `fivepointed stars`, `bricks`, etc. Patterns use the default drawing color, but a specific colors for patterns can be set using `pattern color=`*color*. Some patterns already have color specification included in them. There are no patterns available by default, the library `pgflibrarypatterns` must be loaded first.

• `\shade` is a shortcut for `\path[shade]`, it shades the inside part of the the specified path. The inside is determined according to a chosen rule as in `\fill`. There are three ways of shading, the right one is specified as another optional argument.

`shading=axis` changes hue in vertical direction, so that all horizontal lines have the same color. The default shading is gray on top and `white` on bottom. One can set `top color=`*color*, this not only sets color, but also includes the `shade,shading=axis` option, sets shading angle to 0 and defines middle color as the average of top and the current botom colors. Similarly works `bottom color=`*color*. Option `middle color=`*color* sets the middle color and also the `shade,shading=axis` option, but does not change the rotation angle. It should be used as the last of shading options. `left color=`*color* and `right color=`*color* works just like `top color` and `bottom color`, but they set `rotation angle=90`.
All types of shading are oriented, one can rotate them by setting `shading angle=`*angle* in options.
`shading=radial` starts with gray and gradually (depending on the distance from this center) changes to white. One can change this using `inner color=`*color* and `outer color=`*color*, both of these also set options `shade,shading=radial`
`shading=ball` is obvious, its default color is `blue`, the resulting shade is almost black at the edges and has a white "eye" in the middle of the upper left quadrant. One can change the color using `ball color=`*color*. This basic color is then changed by mixing in white and black. As usual, setting this parameter also sets up `shade,shading=ball`.

• `\shadedraw` is a shortcut for `\path[shade,draw]`, it first shades the inside part of the the specified path and then draws it.

• `\clip` is a shortcut for `\path[clip]`, it uses the specified path to create a mask and outputs of all successive commands (withing the scope where this was used) are cropped by this mask, everything outside is thrown out. If more clips are defined, their intersection is taken as a mask. Clipping does not use any options, and it is advisable not to confuse it by supplying any. This may be unpleasant, for instance if we want to clip and fill time with red color at the same, then we would like to do `\path[fill,clip,fill=red]`, but that would not work. Thus it is better to either do two separate steps (fill, clip), or set `fill=red` before; for instance if before this we have `\scope;` then we can put `\scope[fill=red]`. Note that one can do `\path[fill,clip]` and also `\fill[clip]`.

• `\useasboundingbox` is a shortcut for `\path[use as bounding box]`. The overall dimensions of the complete picture are then determined not by the usual algorithm, but by the specified path (this is then used by TeX to place the picture in text). There are two possible reasons for it. One is that the algorithm does not work all that well with features that extend basic shapes (like arrowheads and extended corners when lines meet), the second possibility is when we want to make a color background and it is nice to extend it a bit further.

When more actions are combined, a logical order is used: first fill, then draw, then clip. However, one can change this somewhat and even reuse one path with changing parameters. Assume that `\path` defines a certain path. If we include another optional parameter (it is a good idea to make it the first one) with `preactions=`{*parameters*}, then the following happens. First, the whole path is constructed virtually. Then, it is drawn/filled etc. according to parameters specified in `preactions`. Third, it is finally drawn/filled etc. as specified. More preactions can be specified and they are performed in the order in which they come. For instance,

```
\draw[preactions={draw,red,line width 4mm}]
  [preactions={fill,fill=red!20}]
  [line width 2 mm,blue] (0,0) circle (1cm)
```

will first draw a red circle, then fill it with pink and finally draw a thinner blue circle, so the red one will be seen as outline. Note that `\draw` is equivalent to `\path[draw]`, so we do no have to specify `draw` in the path specification itself, but we do have to specify it for the first circle, since argument of `preactions` is forwarded to `\path`. Similarly, one can define `postactions`.
Note that the whole path is constructed first, so if we also want to transform the shape when using it, we cannot use coordinate manipulating options like `xshift=1cm`.

3a. Path Specification (Operations)

The shape of a path is determined by operations.

• Typically, a path starts with a **move to** operation, which moves the virtual pen to the specified point, creating a new current location. The format is simple, *point*. This can be used any time in a path specification; whenever TikZ expects an operation and there is a point specification instead, the pen is moved without drawing and a new current location established. In this way we can interrupt the path we are drawing.
Most operations are of the form *operation data* and they all assume that a current location was set first. A typical path therefore starts `\path` *point*, `\draw` *point* etc.
Actions can be chained, so a typical operation creates a new part of the current path and moves the current point to the new location, getting ready for the next operation (if any). Again, exceptions will be noted, and we already encountered one, see relative coordinates above. Operations do not allow for optional arguments, the few exceptional cases will be noted.

• A **rectangular box** is drawn by the operation `rectangle` *point*. The sides are parallel to axes and the current location together with the new supplied point are used as opposite corners.
Example with some options: `\draw[thick,blue] (0,0) rectangle (2,1);`
The option `rounded corners` applies also to rectangles.
Sometimes there is a reason to draw several shapes in one path, then the move-to operation comes handy.

```
\draw[black,fill=red] (-1,-1) rectangle (0,0) (1,1) rectangle (2,2);
```

• The operation `grid` *point* also draws a rectangle, but fills it with lines creating a regular grid. This operation takes an optional parameter `step=` specifying the distance between lines in the grid. One possibility is to set `step=`*dim*, the default value is `step=1cm`. Another possibility is to use `step=`*factor*, then the step in length units is *factor* times the basic length, see Coordinates. The last possibility is to use `step=`*point*, where the grid is set up so that its lower left rectangle has the same lower left corner as the whole grid and *point* determines its upper right corner. As usual, *point* may be also given in polar coordinates.
One can also set up separately `xstep` a `ystep` with values *dim* or *factor*.
Note that when the grid coordinates are not integers, then things get interesting, because grid is always chosen in such a way that it includes the point $(0,0)$.
One often wants the grid to be fainter, there is a predefined style `help lines` that can be used:
`\draw[green,rotate=45] (-2,2) grid[help lines,step=0.5] (2,2);`

• **Circles** are created using `circle (`*radius*`)`, this creates a circle whose center is at the current location and with specified radius.
Example: `\draw[dashed] (1,1) circle (1cm);`
When no dimension is given for the radius, then it is interpreted as centimeters, unless the default unit was redefined.

• **Ellipses** are created using operation `ellipse (`*halfwidth* `and` *halfheight*`)`, the resulting ellipse has its main axes horizontal and vertical and its parameters obviously specify corresponding half-axes.

As of version 3 of TikZ, a new format exists for circles and ellipses. It is a common one, so `circle` and `ellipse` are interchangeable. The new form does not expect radius as data point, but accepts optional arguments. Possible arguments include: `x radius=`, `y radius=`, `radius=` that sets up the x and y radius simultaneously, `at=` that can specify a center directly, and also transformations and many other possible parameters.
`\draw[dashed] (0,0) circle [radius=1cm,rotate=30];`
This in particular allows for the `radius` to be set globally or using the `every circle` style.

• An **arc** is drawn by the operation `arc (`*ang1*`:`*ang2*`:`*radius*`)`. TikZ first determines the center of the arc so that the current location (starting point for the arc) is at angle *ang1* from the center and its distance is *radius*. Then it draws an arc, starting from the current location and turning around the calculated center counterclockwise until it reaches the angle *ang2*. This endpoint then becomes the new current point.
We can also ask for an arc that is a part of an ellipse using the form `arc (`*ang1*`:`*ang2*`:`*halfwidth* `and` *halfheight*`)`.
There are two problems. First, the algorithm sometimes choses the other arc half then the one you want, as it prefers the shorter one. A possible remedy is to draw a longer arc in several parts. Second, the notation is not suitable to situations when we know the center and angles. One possible approach to draw an arc centred at $(a{,}b)$ with radius *rad* between angles *ang1* and *ang2* is
`\draw ([shift=(`*ang1*`:`*rad*`)]`*a*`,`*b*`) arc (`*ang1*`:`*ang2*`:`*rad*`);`
If we use this a lot, it may be worth defining a custom control word in TEX:
`\def\centerarc[#1](#2)(#3:#4:#5);{\draw[#1]([shift=(#3:#5)]#2) arc (#3:#4:#5);}`
As of version 3 of TikZ, a new format is being preferred, where the data is supplied as an argument. Possible parameters are: `radius=`, `x radius=`, `y radius=`, `start angle=`, `end angle=`, and `delta angle=` that is ignored if the previous two are both set.

• The `parabola` *point* operation draws a parabola that has vertex at the current point and moves to *point*. The option `bend at end` places the vertex at the specified endpoint.
We can specify the vertex using the option `bend` *point*. Since this is used often, we can also specify such parabola using the operation `parabola bend` *vertex endpoint*. When this form is used, a special convention is in place for treating relative specification of *vertex*. It is not taken with respect of the current point, but with respect to a point on the segment between the current point and the new point, whose location has to be specified as an optional argument `bend pos=` expecting a decimal number between 0 and 1. Thus if we want to plot a symmetric parabola starting at $(13,0)$ that is 3 units wide and 4 units tall, we could use several codes:

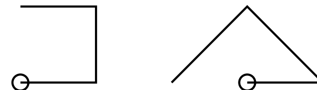
```
\draw (13,0) parabola bend (14.5,4) (16,0);
\draw (13,0) parabola[bend pos=0.5] bend +(0,4) +(3,0);
\draw (13,0) parabola[parabola height=4] +(3,0);
```

• The `sin` *point* and `cos` *point* operations connect the current point with the supplied endpoint using the basic $\left[0,\frac{\pi}{2}\right]$ section of the sine, resp. cosine graph. To do so, TikZ can scale section or flip it vertically, but it does not use rotation.

• The `to` *point* operation draws a **straight segment** from the current location ot the given point. This is the basic operation, and typically is chained. A straight edge is used very frequently, and there is a very convenient shortcut format `--` *point* for it.
`\draw (0,0) -- (1,0) -- (1,0.5) circle (3 pt);`
The `--` operation has a special modification `|-` and `-|` that connects points using horizontal and vertical lines only. For instance, `\draw (0,0) -| (1,1);` will go right then up, while `\draw (0,0) |- (1,1);` will go up then right.
Note that the option `rounded corners` also applies to all joins created by chained `--`, `-|`, or `|-` operations.
Instead of a point one can supply `cycle`, this particular segment is then connected to the first point of the path. Note that there is a difference between `(0,0) -- (1,0) -- (0,1) -- (0,0);` and `(0,0) -- (1,0) -- (0,1) -- cycle;`

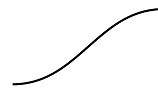
In the first version, the path starts and ends at $(0, 0)$, and the shape at these ends is determined by the parameter `line cap`, see Parameters above. In the second version, the two segments at $(0, 0)$ are connected and special cornering rules apply as specified by `line join` or `rounded corners`.
Note that the `\path` command recognizes when a path was interrupted by a move-to operation and `cycle` refers to the latest start of a continuous part.
Relative coordinates can be very powerful when chaining operations. It is important to keep in mind the difference between the two versions. In both examples below, the current drawing location is being updated at every step, so the segments connect in a natural way, but they differ in the treatment of current referrence location. In the first example it is also being updated, while in the second example the current reference point stays the same.

```
\draw (1,1) circle (3 pt) -- ++(1,0) -- ++(0,1) -- ++(-1,0);
\draw (1,1) circle (3 pt) -- +(1,0) -- +(0,1) -- +(-1,0);
```



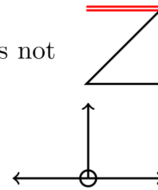
Compare this with a third version in section on the `edge` operation.

Unlike the `--` shortcut, the full version `to` *point* accepts optional parameters that curve the connecting segment. The basic options are `bend left` and `bend right`, where the left and right are based on the view from the current point towards the target point. The trick is that when `bend right` is used, the curve actually starts more to the right than it should and then bends left towards the target point, and `bend left` is similarly counter-intuitive. We can influence the intensity of bending using `bend right=`*angle* or `bend left=`*angle*, where the *angle* is the difference between the outgoing and ingoing angle. This bending angle can be set up globally with the `bend angle=`*degrees* option.
These two angles can be controlled directly using options of `to`. The option `out=`*angle* specifies at which angle the edge should start, similarly `to=`*angle* specifies the angle of the ending connection and the edge creates an arc of radius determined by TikZ.
When the `out-in` pair does not allow one arc to be constructed, then TikZ creates two bends. For instance, to make `\draw (0,0) to[out=0,in=180] (4,2);` work, the curve starts right, then bends up, then bends right and reaches $(4, 2)$ in a horizontal direction.



• Another way to draw segments is the `edge` operation. The outcome looks like the outcome of the `to` operation, but it is reached in a different way. All the previous path operations (and those that follow) create segments that immediately become integral parts of the resulting path. The `edge` operation creates a segment, but keeps it separate, and it is added to the path only after the whole path is constructed and drawn.
This has two important consequences. First, it allows `edge` to accept parameters not only of the `to` operation, but also parameters that are otherwise applicable only to the whole path. We can thus change color and shape of individual segments, add arrowheads (see below) etc. Second, since the operation is not an integral part of the path, the current locations for drawing and reference are not updated. This can be troublesome when chaining segments.
Example: `\draw[thick] (0,0) -| (1,1) edge[double,color=red] (0,1) -- cycle;`
Note that we did not obtain a square when we closed the path, because the segment leading to (0,1) is not "seen" by the path constructor.
But sometimes this can play to our advantage. We will show a third version of the command that showcased relative coordinates:

```
\draw (1,1) circle (3 pt) edge[->] ++(1,0) edge[->] ++(0,1) edge[->] ++(-1,0);
```



Sometimes this is exactly what we want.

• The `..` operation or **curve to** operation draws curves. The operation `..controls` *point1* `and` *point2* `..` *point3* creates a cubic Bezier curve connecting the current point (call it *point0*) with *point3* (which becames a new current point) using a curve in such a way that the direction from *point0* to *point1* gives the starting direction of the curve and the direction from *point2* to *point3* determines the direction from which the curve enters *point3*.
The intensity of the influence (how much is the resulting curve forced to follow this direction) is determined by the mutual distance of the determining points. If *point2* is not given, it is assumed that *point2*=*point3*. For instance,
`\draw (0,0) ..controls (0,1) and (4,1) .. (4,0);` will do something like the top of an ellipse, while
`\draw (0,0) ..controls (0,1) .. (1,1);` will do exactly a quarter of a circle.
These operations can be chained to create longer Bezier curves. The joins between these curves, and between them and edges, again follow the `line join` or `rounded corners` rules.

• The `plot` operation creates a curve by connecting points supplied in specification that follows. The curve is constructed and then it becomes a part of the path. This means that the location of the current point is not taken into account. The operation `--plot` is almost identical, but connects the first supplied point with the current point. Both versions make the last supplied point into a new current point.
There are three ways to supply data. The operation `plot coordinates`{*point1 point2* ...} connects the specified points.
The operation `plot file`{*file*} loads points from the *file*. It accepts text files where on every line there are two numbers separated by a space (defining a point), further characters after these two numbers on the same line are ignored with two exceptions. If the point data are followed by `o`, it is considered an outlier and usually a new subpath starts. If the point data are followed by `u`, it is considered to be undefined and usually a new subpath starts. Empty lines and lines that have `%` or `#` as the first character are also ignored, the latter are used for comments.
The operation `plot (`*coordinate expression*`)` calculates points to connect using the expression supplied in *coordinate expression*. The variable is `\x` by default, but we can change it using the option `variable=`*var* of `plot` to any other string of letters preceded by a backslash. For instance,

```
\draw[scale=0.5] plot[variable=\t] ({\t},{\t*\t});
```
produces the graph of parabola in Cartesian coordinates. Note that the operation `plot` can take options, but only those related to the construction of the curve, others have to be used with the `\path` command. Note also the formulas. It is recommended to enclose both coordinate specifications in groups to prevent problems, namely expressions that feature paranthesis would make TikZ think prematurely that the specification ends. Most elementary functions are available, including `abs` and `sqrt`. Powers do not work as expected, for instance `\t^2` in fact produces $\mathrm{sgn}(t)t^2$, so we used multiplication instead.
Note that the range of values is limited in TeX, which applies even to intermediate steps in calculations, this has to be taken into account when writing the expressions. Finally, note that the general setting is that of a parametric curve, we can supply a formula for both coordinates. This allows for easy scaling, drawing of inverse functions and other tricks.
The values of the variable are determined by two options of `plot`. The range for the variable is defined using `domain=`*n1*`:`*n2*, it can be done globally (for more graphs) If this is not specified, the default is `domain=-5:5`. The number of points that are to be taken from this domain is defined using `samples=`*number*, default value is `samples=25`. It is also possible to specify directly sampling values using `samples at={`*n1*`,`*n2*`,...}`. If by some mistake both specifications `domain/samples` and `samples at` are used, then only the last defined is used. By the way, `\foreach` can be used in `samples at` specification.
We could draw a circle like this: `\draw plot[domain=-180:180,samples=180] ({1.3*cos(\x)},{1.3*sin(\x)});` note that the angle is taken in degrees by default. If we want some variable to be treated as radians, we follow its reference with `r`: `\draw plot[domain=-pi:pi] ({1.3*cos(\x r)},{1.3*sin(\x r)});`

We can control how the plot is constructed using further **options** of `plot`, but they can be also included in `\path` (or `\draw` and equivalents) if we want them to apply to more `plot` segments in the path. Implicitly, data points are connected using straight segments, but one can change this using optional arguments. `sharp plot` is the same as the default behaviour, `smooth` uses a smooth curve, it also has a version `smooth cycle` that closes the curve (no need to specify the first point at the end of the list again) and handles properly the connecting point. The rounding algorithm works best if bending angles are smaller. The amount of rounding can be specified by the optional argument `tension=`*number*, where 0 does no curving at all (like `sharp plot`), while the value 1 does maximal bending,
```
\draw[thick,red] plot[smooth cycle,tension=1] coordinates{(0,0) (1,1) (0,2) (-1,1)};
```
will draw a circle. The default value is `tension=0.55`.

Marks can be added to a plot using the option `mark=*` (does small black filled circles), `mark=+` (does small black crosses), and `mark=x` (does small black x-oriented crosses). More marks are available when the `pgflibraryplotmarks` library is loaded. Marks are put at every point used in constructing the graph, we can ask for some of them to be skipped using `mark repeat=`*number1*, then every *number1*-th mark will be drawn. The phase at which this starts can be determined by `mark phase=`*number2*, so marks at points *number2*, *number2*+*number1*, *number2*+2·*number1* etc. will be plotted. Another possibility is to use `mark indices={`*n1*`,`*n2*`,...}`. One can also influence the way marks are drawn, one option is direct, `mark size=`*dim* will change its size, more options (color, rotation, scaling) can be set using `mark options={...}`.
```
\fill[fill=blue!20] plot[smooth,tension=0.7,mark=x,mark options={color=blue,rotate=20}]
    coordinates{(0,0) (1.57,1) (3.14,0)};
```
The option `only marks` will draw only marks, not the curve.
```
\draw[red] plot[samples=9,domain=-2.0:2.0,mark=*,only marks,mark size=1.5pt] (2*\x,{abs(\x)^2});
```
We made the graph flatter by scaling the x-axis, and used another trick to obtain the square with proper sign. The multiplication approach is actually preferred when possible.

The option `const plot` connects data points using vertical and horizontal segments, creating a step function. For further modifications see the TikZ Manual.

The option `ycomb` does not connect the supplied points with segments mutually, but connect each with with the x-axis using a vertical line. In this way we construct a column graph. Similarly, `xcomb` connects with the y-axis using horizontal segments. `polar comb` connect the marks with the origin.
```
\draw[line width=8mm] plot[ycomb] coordinates{(1,3) (2,7) (3,5)};
```
The `comb` options can be also specified in `\path` or globally.
If we also specify marks, then we get something that looks like a fence with little balls on tops of planks.
```
\tikzpicture[ycomb]
\draw[color=red,line width=6pt] plot[mark=*] coordinates{(0,1) (1,1,5) (2,0.75)};
```

The point specifications can be also done in other coordinate systems. For instance, a spiral can be done in polar coordinates. `\draw[smooth] plot[domain=0:6.283,variable=\r] ({\r r}:{\r/2});`
We had to indicate that the angle is in radians. We could have also tried something like
```
\draw[scale=0.5,smooth] plot[domain=0:720,variable=\r] ({\r}:{\r*pi/360});
```

• The `SVG` operation specified as `svg` *data* creates a curve based on SVG path specification that can be supplied in quotations marks or as a group. The basic unit is a point, but it can be changed using the `scale=` argument of `svg`.
```
\draw[thick] svg[scale=1cm] "M 0 0 L 2 0 H 2 Z";
```
Note that the `a` and `A` operations are numerically unstable.

3b. Decorations

The general look of a path can be influenced by the parameters covered in chapter 2 Parameters, and these are often used both globally (as options of `\tikzpicture` or `\scope`) and locally as options of `\path` or `\edge`. Here we look at two further modifications that are typically used locally and fall into a more general category of decorations.

The `double` option actually first draws a very thick segment using the default color and then draws a narrow segment over it using another color, by default it is `white`, but one can change it: Instead of `double` one uses `double=`*color*. The width of this fill-in can be changed using `double distance=`*dim* (both parameter specifications can be done globally).

Probably the most useful decoration is **arrowheads**. These can be placed at the beginning or end of a path using the option `arrows=`*beg*`-`*end*. The part `arrows=` may be left out. Specification *beg* or *end* may be empty, then there is no arrowhead. There are actually two basic kinds, a true arrowhead marked by `<` or `>` and a short perpendicular bar marked by `|`. Example: `|->`. The actual type of the arrow can be changed by a general option setting, see Parameters above. If we want to change the type locally, we need not do it using an option, this can also be done by including the proper name in the arrow specification, for instance, we would write `-stealth` instead of changing the default type and writing `->`.
The actual arrows can be doubled or combined with `|` as well, e.g. `\draw[|<->>]`, and we obtain the corresponding picture. However, for these double arrows the trick with direct naming of type like `latex latex` does not work, that would have to be done by changing the default type. Some examples:

`draw[->]`   `draw[|-<]`   `draw[-to reversed]`   `draw[|-latex]`   `draw[<->>]`

The arrowhead always ends at the specified location, so the line that leads to it is shortened if necessary. We can also make the arrowhead end sooner using the `shorten` option, globally or locally.
The size of arrows depends on the thickness of the underlying line, it cannot be specified directly. If we are not happy with this default, there is a trick: We draw the segment at the desired thickness, then overlay it with a very short segment with an arrow and set the thickness of this segment so that the arrow has the right size.
We can attach an arrowhead to just one segment of the path if we supply it as an option of the `edge` operation. However, then we have to take into account that the current point is not being updated. We can do that manually using a move-to operation, but then we may run into trouble with line joining.

```
\draw[red] (0,0) -- (1,0) edge[blue,<->] (1,1) (1,1) -- (0,1);
```

More substantial decorations can be obtained using the option `decorate`. However, for that one first has to load the `decorations` library, see the TikZ manual.

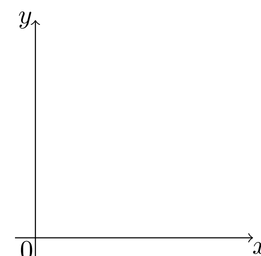
4. Placing text

Text can be placed in a picture as a node. It has two forms. As a command `\node` {*text*} it can be used independently. As an operation `node` {*text*} it can add some text to a path. This is done after the path is constructed, just like the `edge` operation, and thus allows one to locally change options.
By default, the text is placed centered at the current location. However, we can also specify the placement location using the form `node at` *point*.

```
\node at (3,1) {Something};
```

In this way we can add labels to graphs, coordinate axes etc. For instance, we can set up the Carthesian system using

```
\draw[->] (-0.3,0) -- (3.2,0);
\node at (3.3,-0.15) {$x$};
\draw[->] (0,-0.3) -- (0,3.2);
\node at (-0.15,3.2) {$y$};
\node at (-0.13,-0.18) {$0$};
```



Note that when we scale a picture using `scale`, then this only transforms coordinates, but the size of the text stays the same (which makes sense, we do not want labels to become tiny). Consequently, we usually have to modify placement after scaling a picture.

The argument can contain text including inline math, but we should avoid displayed math. We should also keep in mind that we cannot freely use words that serve as options for TikZ. And when we do have to, we need to tell TikZ that they are to be taken literally, not interpreted. This is done by preceding them with `\verb!`.

```
\node at (0,0) {This is \verb!node taken literally}
```

We can also influence treatment of the text. By default, the text is set as a line, broken if it exceeds the page width. We can restrict the width of the resulting text block using the option `text width=`*dim*. If the text is shorter, the size of the node will be taken larger as specified. When the text is broken, we can specify the treatment of line ends. By default the ragged right border is used. One can change it using options `text justified` (adjusts to equal length lines), `text ragged` (adjusts reasonably and leaves the right border ragged where necessary), `text badly ragged` (no space adjusting at all), `text centered` (it also adjusts spaces to make lines at least approximately equally long), and `text badly centered` (no space adjusting).
As of version 3 of TikZ, we can use instead the the option `align=` with possible values `left`, `flush left`, `center`, `flush center` (wich is recommended over `center` as it does nto try to mess with spaces), `right`, `flush right`, `justify`, and `none`.
We can break a line any time manually with \\.

We can also influence font, for instance `font=\itshape` puts text in italics, one can also try `\rmfamily`, `\bfseries`, `\sffamily` (sans serif), `\scshape` (small caps), and `\normalfont`. We can also use `font` to specify size, say, `font=\footnotesize`, other interestiong values may be `\small`, `\tiny`, `\small`, `\normalsize`, `\large`, `\Large`, `\huge` and more.
The text is printed in the default `draw` color. We can set it using `text=`*color*. We can also use `\node[red]`, but then `red` is assigned as a `color`, so it also applies to fills and draws and we may not want that, see below.
One can also specify `text height=`*dim* and `text depth=`*dim* to influence the virtual size of the text box (just like in TEX).

The text we are placing is always enclosed in a virtual shape. There are two types that we can choose using the option `shape=`*shape*. The default is `rectangle` and the other possibility is `circle`. However, TikZ also accepts just `circle` or `rectangle`. This virtual shape influences placement of the contents of the node and position of connecting lines (see below), but we can also show it by choosing an action with `draw` and/or `fill`. These two options can also be used to specify the color of drawing/filling, and other options may be used.

```
\node[circle,blue,draw,fill=green!20,text=red,thick] at (0,0) {a};
```

This will draw a small blue circle with green filling containing a red “a” inside. Note that `color=`*color* or `\node[`*color*`]` sets the color for all three components: text, shape, and fill.
The size of circles and rectangles is determined by the size of text plus a little offset that can be set with the option `inner sep=`*dim*. We can also influence the size of the shape by setting `minimum height=`*dim* and `minimum width=`*dim*, or the general specification `minimum size=`*dim*. All this can be done globally in the argument of `\tikzpicture`.
When drawing these shapes, appropriate options apply, for instance `double`, or `rounded corners` for the rectangle.
If the same options are used with more node specifications, it may be easier to set up styles (see Styles) in the optional argument of `\tikzpicture`. This can be especially powerful when coupled with the possibility to supply an empty argument, which offers a simple way to draw circles and rectangles at specific locations.

```
\tikzpicture[inner sep=2 pt,minimum size=3pt,mynode/.style={shape=circle,draw=blue,fill=red!20}]
\node[mynode] at (13,13) {};
```

There are also three special styles, `every node`, `every rectangle node`, and `every circle node` that are installed every time we start a node of the right type.

```
\tikzpicture[every circle node/.style={draw,fill=red}]
\node[circle] {$x$};
```

The style approach is very powerful and very advanced tricks are available. Here is a useful one. It needs a library loaded with `\usetikzlibrary{calc}`, then we can do

```
\tikzpicture[cancel/.style={path picture={
\draw[#1] ($ (path picture bounding box.south west)+(-3pt,6pt)$)
    -- ($(path picture bounding box.north east)+(3pt,-4pt)$); }}]
\node[inner sep=3pt,cancel=red] {$2x+3=y$};
\endtikzpicture
```

which prints $2x+3=y$

The argument can be moved away from the target point by optional arguments `left`, `right`, `above`, `below`. If more are used, then only the last one takes effect (so one cannot do tricks like `above,left`), but fortunately there are also options `above left`, `above right`, `below left`, and `below right`. The default distance can be changed by using the above code words as parameters, for instance `below=3pt` or `above right=5pt`. Thus we could set up the x-axis also like this:

```
\draw[->] (-0.5,0) -- (3.2,0);
\node[below] at (3.2,0) {$x$};
```

Perhaps we would prefer to shift x a bit to the right, but `below right` is too much. We can use the usual adjustment parameter:

```
\node[below,xshift=2pt] at (3.2,0) {$x$};
```

There is an alternative that works a bit counter-intuitively, but it has its uses. The option `anchor` specifies which part of the argument as a box should be positioned at the target location. So instead of `node[left]` we would use `node[anchor=east]`; the middle of the right border of the argument will be placed at the target location, so the argument itself is to the left. The eight basic directions including `south east` etc. are available.

One can put **labels** next to objects positioned by `node`. They are specified as another optional argument `label=`*position*`:`*text*, for instance to draw a circled “a” with a label we could do

```
\node[circle,draw,label=above:$a\ge2$] at (1,0) {a};
```

The eight positons mentioned above are available here again, or we can put in a number that specifies angle in degrees. However, note that now the `below=1pt` trick will not work, the distance is controlled by parameter `label distance=`*dim* that can be set both locally and globally.
It is a good idea to enclose the label specification as a group, it helps readability, and also allows to add an optional argument for the label, typically a color specification:

```
\node[label={[red],60:$A$}] at (1,0) {};
```

This A will be moved in the direction 60 degrees from $(1,0)$ and printed in red. Note the empty argument. If all labels are to have the same color, it is easier to set it globally by putting `every label/.style={red}` in the optional argument of `\tikzpicture`.
In effect, `\node[below] at (0,0) {$x$};` and `\node[label=below:$x$] at (0,0) {};` do analogous jobs.
Note that there is also an option `pin=` that works exactly like `label=` but the resulting label is connected to the node that is being labeled by a short segment. Instead of `label distance` it uses `pin distance`. Pin takes an optional argument `pin edge` specifying line thickness and color, arrows and possibly also decoration.

```
\node[circle,draw,pin={[pin distance=0pt,pin edge={<-,thick,red}]left:start}] at (0,0) {};
```

The operation version `node` can be inserted into a path. Typically it is put after a point specification, and it uses the latest current point as the target position. This offers another way to set up the x-axis.

```
\draw[->] (-0.5,0) -- (3.2,0) node[below,xshift=2pt] {$x$};
```

Recall that `edge` does not update current locations, so if we do the arrow as `\draw (-0.5,0) edge[->] (3.2,0)` then the x would actually appear below the left end.
On the other hand, the `plot` operation updates the current location, so we can conveniently draw a graph with a label like this:

```
\draw[thick] plot[domain=0:4,smooth] ({\x*\x},{\x}) node[right] {$f(x)=\sqrt{x}$};
```

Just like `edge` segments, the object created by `node` is only added to the path after it is constructed, therefore it can have local properties, otherwise it inherits the parameter settings (color etc) from the parent `\path` specification.

We can also insert a `node` after a `to` operation (also in the `--` form), `edge` operation and the curve to `(..)` operation, and then the label is attached to the segment. By default, the point in the middle of this segment is taken as the target point for the node, but we can change this using the option `pos=`*factor*, where *factor* is a number between 0 (the beginning) and 1 (the end). Some locations are predefined and we can refer to them as `at start`, `very near start` (`pos=0.125`), `near start` (`pos=0.25`), `midway` (default), `near end` (`pos=0.75`), `very near end` (`pos=0.875`), and `at end`.
However, this places the `node` object directly on the line, so we will most likely want to use `above`, `right` etc. as well. Another possibility is to use `auto`. We can actually define the meaning of this as a parameter (for instance globally), the default is `auto=left`, but now not in the sense of spatial orientation (west), but one of the eight basic directions is chosen so that the displacement is as close to perpendicular to the segment as possible to the left from the point of view of the path. The other possibility is `auto=right`. If we want the other side, we can use the option `swap`. For instance,

```
\draw[->] (0,0) to node[pos=0.2,auto] {$f(x)$} node[pos=0.2,auto,swap] {1-1} (2,1);
```

will draw an arrow, it will have "$f(x)$" on its left side and "1-1" on its right side of the curve, closer to its start. Which shows that more nodes can be inserted.
When putting text around a curved edge, we may want it to be positioned parallel to it. This is done by the option `sloped`. Example:

```
\draw (0,0) to[out=0,in=-135] node[near start,sloped,above] {near start} (4,2);
```

It is also possible to specify location in a `node` operation, as in

```
\path[draw] (0,0) -- (2,0) node[mynode] at (1,1) {13} -- (2,1);
```

but then there is not much point in having this node as a part of the path, we can set it directly using `\node`.

There is an exception ot the inheritance of properties from `\path`. Since the `node` objects are added only after a path has been constructed, the transformations do not apply to them. This can be sometimes awkward, and the option `transform shape` makes sure that this node is transformed just like the rest of the path. On the other hand, it also means that we can specify transformations for a specific node and they do not get composed with outside transformations. For instance,

```
\path[rotate 30] (0,0) node[transform shape] {$A$};
\path[rotate 30] (1,0) node[xshift=2cm,rotate=60,scale=2] {$B$};
```

means that A will be rotated by 30 deg, while to B only the three transformations specified in its `node` will apply.

As noted above, scaling of picture does not influence the size of the text placed by a node and decoration about it if present. If we do want to scale it, we can put `scale=` as an argument of a particular node. If we want to scale the whole picture including all nodes, we use the `every node` style:

```
\tikzpicture[scale=0.7, every node/.style={scale=0.7}]
```

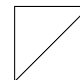
We can also use `every node/.style={transform shape}` that applies all global transformations to nodes, but that would also include rotations and perhaps we do not want to rotate our texts.

5. Referring to locations and objects

One of the useful features of `node` is that we can name the objects that we create with them, and then refer to them by name. There are two ways to name a node. One is using the `name=`*name* option, where *name* cannot contain special characters, commas, periods etc.; it is easiest just to stick to letters, numerals, spaces, hyphens and underscores. Another possibility is to supply the name in parantheses immediately after `node` and its optional parameters. The following two codes create a reference `mypoint` for the point $(13,1)$: `\node[red,name=mypoint] at (13,1) {a}` and `\node[red] (mypoint) at (13,1) {a}`.

If we just want to name a location, we can use an empty argument. In this way we can also name points used within a path construction: `\path (13,1) node (mypoint) {} -- (7,3);` For this purpose there is a special command `coordinate`, so we can do `\path (13,1) coordinate (mypoint) -- (7,3);` We can also assign a name directly using `\coordinate (`*name*`) at` *point*.

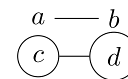
A named location can be used in place of any *point*.



```
\draw (0,0) coordinate (mypoint) -- (1,1) rectangle (mypoint);
```

This is actually a convenient shortcut, the proper format for the reference is `(node cs:name=`*name*`)`.

When an object is created by `node`, an imaginary shape is created around it even if we do not actually draw it. When we name such nodes and connect them, the connecting curve is shortened and reaches only to the borders of these shapes.




```
\node (a) at (0,0.5) {$a$}; \node (b) at (1,0.5) {$b$}; \draw (a) -- (b);
\node[circle,draw] (c) at (0,0) {$c$}; \node[circle,draw] (d) at (1,0) {$d$};
\draw (c) -- (d);
```

We can influence the location of the point where connecting curves attach to these boxes. When a node is named, eight anchor points are created on the perimeter of its bounding box in compass directions. The references for these anchors of a node *name* are *name*`.north`, *name*`.north east`, *name*`.north west`, *name*`.east`, *name*`.west`, *name*`.south`, *name*`.south east`, and *name*`.south west`. The original reference *name* is also available as *name*`.center`.

```
\node[circle,draw] (point1) at (0,0) {$a$};
\node[rectangle,draw] (point2) at (1,0) {$b$};
\draw[-latex] (point1.north east) to[bend right] (point2.north west);
```

If this does not work well enough, we can actually specify a precise connecting position using angle position. The format is `angle=`*angle* and we need to use the full format for node reference: `(node cs:name=mypoint,angle=60)`.

There is a special node with compass anchor points available. `current bounding box` refers to the whole picture created by the `\tikzpicture`.

Calculating references.

It is possible to combine coordinates (both explicit and referenced) into linear combinations. The general form is `($`*formula*`$)`, for instance, `($0.5*(point1)+0.5*(point2)$)` refers to the point exactly between the two coordinates defined above. The implementation not only interpolates, but also extrapolates, for instance it can handle expressions like `($1.5*(point1)-0.5*(point2)$)`.

When processing the formula at the start or after `+` or `-`, TikZ expects to see `(`. If some other character is encountered, TikZ starts processing it as a multiplicative factor, and includes into it everything until `*(`. This may not be convenient, for instance if the factor also includes some calculation that features multiplication of a sum, and thus it is a good idea to enclose multiplicative factors into groups. For instance, `(${3*(1+sin(1))}*(3,2)+{1/3}*(mypoint)$)`.

Since weighted averages are used often, there is a special format for them: `($`*point1*`!`*factor*`!`*point2*`$)`. Note that the *factor* refers to the second point, while the first point is multiplied by (1 − *factor*). Only factors between 0 and 1 are allowed.

There is a special modification `($`*point1*`!`*factor*`!`*angle*`:`*point2*`$)`. Here the segment between the points is constructed, then rotated by the specified angle about *point1*, and then the location is identified on this rotated segment.

These "partway modifiers" can be chained. `($`*point1*`!`*factor1*`!`*point2*`!`*factor2*`!`*point2*`$)` means that first, a supplementary *pointx* is determined based on `($`*point1*`!`*factor1*`!`*point2*`$)`, and then the definitive location is determined as `($`*pointx*`!`*factor2*`!`*point2*`$)`.

Instead of a *factor* we can supply a dimension (including a unit), then the point is constructed on the connecting segment at the specified distance from the first point. Say, `($(mypoint)!2cm!(13,0.7)$)`. Again, an angle of rotation can be present.

Finally, when we supply yet another point instead of a factor, `($`*point1*`!`*pointP*`!`*point2*`$)`, then a location is determined by an orthogonal projection of *pointP* onto the segment connecting *point1* and *point2*.

6. Programming

A path specification may be stored using the usual TEX definition way:

```
\def\triangle{ -- +(1,0) -- +(0,1) -- cycle}            note no semicolon here
\draw[red] (13,5) \triangle;
\fill[grey] (-1,7)\triangle;
```

We can program loops in a `tikzpicture`. The format is `\foreach` *variable* `in` {*set*}. The variable could be just like for the `plot` operation, that is, a backslash followed by a string of letters. The *set* describes values, typically numbers, but it can be also strings, TEX commands, or names of named objects.

The loop applies to one command that follows, or to a group of commands delimited by braces {}.

Example:

```
\foreach \x in {0.5,1,2} \draw (\x,1) circle (5mm);
```

If we want the variable to run through a finite equally spaced set of numbers, we use the three-dot notation $\{a,\ldots,b\}$ for numbers $a < b$, the variable then goes by increments of 1 and stops if it would exceed b. For instance, with the specification `{1.3,...,5}` the last number would be 4.3. For $a > b$ it goes down by 1.

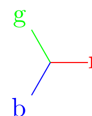
If we specify $\{a, A, \ldots, b\}$, then the difference between a and A is used as a step. This does not always work well when successive ranges are used, e.g. `{1,...,4,6,...,10}`, so more numbers may be needed to set the step size right: `{1,...,4,6,8,...,10}`.

Cycles can be nested, in which case the next `\foreach` is interpreted as the command of the previous one, so we do not have to put it inside a group:

```
\foreach \x in {1,...,4}
  \foreach \y in {1,...,4}
    {
      \draw[color=blue,fill=red] (\x,\y) circle (3mm);
      \node[below] at (\x,\y) {$(\x,\y)$};
    }
```

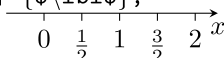
The loop command also allows for several variables in a synchronous loop, the variables and values are separated by a slash.

```
\foreach \angle/\lbl/\clr in {0/r/red,120/g/green,240/b/blue}
    \draw[\clr] (0,0) -- +(\angle:5mm) node[label distance=-7pt,label={\angle:\lbl}] {};
```



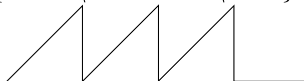
If the second specification is missing, the first one is taken. Here is how one can set up the x axis with labeled tick marks.

```
\draw[->] (-0.5,0) -- (2.3,0) node[below] {$x$};
\foreach \x/\lbl in {0,0.5/\frac12,1,1.5/\frac32,2}
    \draw (\x,0) -- (\x,-0.1) node[below] {$\lbl$};
```



A cycle can be placed inside a `\path` specification. We can create a saw using

```
\draw (0,0) \foreach \x in {1,...,3} { -- (\x,1) -- (\x,0)} -- (4,0);
```



So much for the basics. To unlock the full potential of TikZ, consult the 1300+ page manual.