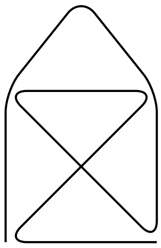
Part I

教程和指导

by Till Tantau

为了帮你入门 Ti*k*Z，本手册没有立刻给出长长的安装和配置过程，而是直接从教程开始。这些教程解释了该系统所有基本特性和部分高级特性，并不深入所有细节。这部分还指导你在用 Ti*k*Z 绘图时，如何继续前进。



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

1 教程：给 Karl 的学生的一张图

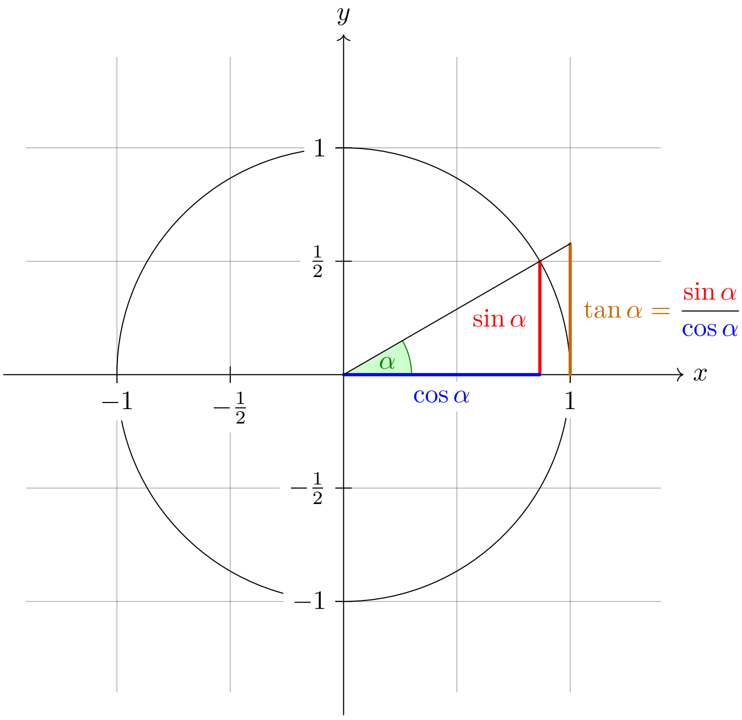
该教程是写给 Ti*k*Z 的新手的，它并不详细罗列 Ti*k*Z 的所有特性，而是只讲那些你可能立马会用到的。

Karl 是一名高中数学和化学老师，他以前用 LaTeX 的 `{picture}` 环境，为习题和考试画图。尽管结果可以接受，但是绘图通常要花很久，并且还有一些问题，比如线之间有微小的角度错误，圆形好像也很难画对。当然，他的学生并不关心线之间的角度对不对，因为 Karl 出的试题太难了，画得再好也没用。但 Karl 对绘图的结果从不满意。

Karl 的儿子对绘图结果更不满意（毕竟他不要考试），儿子告诉 Karl 他也许愿意试试一个新的宏包来绘图。令人困惑的是，这个宏包似乎有两个名字：首先 Karl 需要下载和安装一个叫 PGF 的宏包，接着他发现这个宏包里还有另一个宏包，叫 Ti*k*Z，应该代表“Ti*k*Z ist *kein* Zeichenprogramm”（Ti*k*Z 不是一个画图程序）。Karl 觉得有点奇怪，从 Ti*k*Z 的名字来看，似乎并不能满足自己的要求。不过，由于他用过一段时间的 GNU 软件，并且知道“GNU’s Not Unix!”（GNU 不是 Unix）这句典故，因此似乎有点希望。他儿子告诉他，Ti*k*Z 起这个名字，是为了提醒人们，Ti*k*Z 不是一个用鼠标和平板画图的程序，相反，它更像是一门“图形语言”。

1.1 问题陈述

Karl 想在下次学生习题中放一张图，他现在正在教正弦和余弦。他想得到这样的图（理想情况）：



在本例中，角 α 为 30°（等于 $\pi/6$ 弧度），α 的正弦，也就是红色线段的高度，为

$$\sin\alpha = 1/2.$$

根据勾股定理，我们有 $\cos^2\alpha + \sin^2\alpha = 1$。因此蓝色线段的长度，也就是 α 的余弦，肯定为

$$\cos\alpha = \sqrt{1-1/4} = \sqrt{3}/2.$$

由此可以得到 $\tan\alpha$，也就是橙色线段的高度，为

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{3}.$$

1.2 设置环境

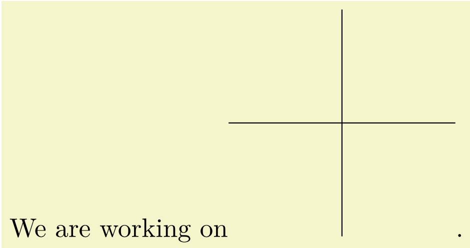
要在 Ti*k*Z 中画一张图片，你需要在图片开头告诉 TeX 或者 LaTeX 你想开始画一张图。在 LaTeX 中需要用 `{tikzpicture}` 环境，在 plain TeX 中你只要用 `\tikzpicture` 开始图片，再用 `\endtikzpicture` 结束图片。

1.2.1 设置 LaTeX 中的环境

Karl 是一个 LaTeX 用户，所以他的设置是这样的：

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

当程序执行时，也就是用 `pdflatex`，或者是 `latex` 加上 `dvips`，输出包含这样的内容：




```
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
```

诚然，图还算不上完整，不过我们已经建好坐标轴，还画好组成坐标轴的线了。Karl 突然有点沮丧，似乎离画完还有不少距离。

让我们更仔细地看一下代码。首先导入 `tikz` 库，也就是基础 PGF 系统所谓的“前端”。本手册还会提到基本层，更基础也更难用。这个前端语法更简单，让画图变得更容易了。

里面有两句 `\draw` 命令，意思是：“从此处到逗号间指定的路径，需要画出来。”第一条路径指定为 `(-1.5,0) --(0,1.5)`，意思是：“一条从点 $(-1.5, 0)$ 到点 $(-1.5, 0)$ 的线”。在这里，位置是用一个特殊的坐标系统指定的，初始的单位长度是 1cm。

Karl 很高兴地注意到，环境中已经预留了足够的空间来容纳图片。

1.2.2 设置 Plain TEX 中的环境

Karl 的妻子 Gerda，恰好也是一名数学老师，她不是 LATEX 用户，而是用 plain TEX ，因为她更喜欢“老派的方法”。她也能用 Ti*k*Z，不过不是 `\usepackage{tikz}`，而是 `\input tikz.tex`，同样，`\begin{tikzpicture}` 和 `\end{tikzpicture}` 也应该换成 `\tikzpicture` 和 `\endtikzpicture`。

因此她应该用下面的语句：

```
%% Plain TeX file
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda 既能用 `pdftex` 也能用 `tex` 加 `dvips` 来排版这个文件。Ti*k*Z 会自动识别她用了哪个驱动。如果她想用 `dvipdfm` 结合 `tex`，她既可以选择修改 `pfg.cfg` 文件，也可以选择在导入 `tikz.tex` 或是 `pgf.tex` 之前加上 `\def\pgfsysdriver{pgfsys-dvipdfm.def}`。

1.2.3 设置 ConTEXt 中的环境

Karl 的叔叔 Hans 用 ConTEXt，Hans 也能像 Gerda 一样使用 Ti*k*Z。他需要把 `\usepackage{tikz}` 换成 `\usemodule[tikz]`，同样，`\begin{tikzpicture}` 和 `\end{tikzpicture}` 也要换成 `\starttikzpicture` 和 `\stoptikzpicture`。

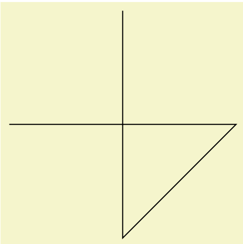
他的样例代码像这样：

```
%% ConTeXt file
\usemodule[tikz]

\starttext
  We are working on
  \starttikzpicture
    \draw (-1.5,0) -- (1.5,0);
    \draw (0,-1.5) -- (0,1.5);
  \stoptikzpicture.
\stoptext
```

Hans 现在就能像平常一样，用 `texexec` 或者 `context` 来排版这个文件了。

1.3 直线路径创建

Ti*k*Z 里所有图片的基本单元是路径（path）。路径就是相连的直线和曲线（并不是整个图片都是如此，不过让我们暂时忽略复杂的部分）。你指定一个起点作为路径开始，圆括号中为点的坐标，比如 `(0,0)`。紧接着是一系列的“路径扩展操作”，最简单的是 `--`，我们之前用过，它后面必须跟着另一个坐标，它将原来的路径沿直线延伸到新的位置。比如，如果我们打算把坐标轴的两条路径并成一条，那么会得到下面的结果：



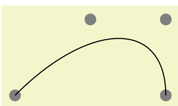
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl 有点困惑，因为这里没见到 `{tikzpicture}` 环境，而是用了一个小命令 `\tikz`。这个命令既可以接受单个参数（放在花括号中，比如 `\tikz{\draw (0,0) --(1.5,0)}` 会生成一条线 ），也可以将其放到 `{tikzpicture}` 环境中，每句用以分号结尾。一般来说，所有的 Ti*k*Z 绘图命令必须作为 `\tikz` 的参数，或者放在 `{tikzpicture}` 环境里。幸运的是，`\draw` 命令只定义在这种环境下，因此你很少会有出错的机会。

1.4 曲线路径创建

Karl 下一步想做的就是画圆，很明显用直线做不到，所以我们需要一些方法绘制曲线。为此，Ti*k*Z 提供了一个特殊的语法，需要用一到两个“控制点”。它背后的数学并不简单，但是基本思想是：假设你位于点 x，第一个控制点是 y，那么曲线开始时会沿着从 x 到 y 的方向，也就是说，该曲线在点 x 处的切线经过 y。接着，假设曲线的结束点为 z，第二个控制点是 w，那么该曲线就会在点 z 处结束，并且曲线在点 z 处的切线经过 w。

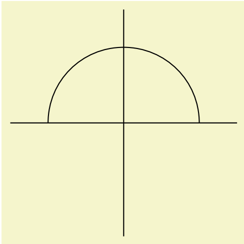
例子如下（为了更清楚地说明，这里画上了控制点）：



```
\begin{tikzpicture}
  \filldraw [gray] (0,0) circle [radius=2pt]
                   (1,1) circle [radius=2pt]
                   (2,1) circle [radius=2pt]
                   (2,0) circle [radius=2pt];
  \draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

以曲线方式延伸路径的通用语法是 `.. controls` ⟨第一个控制点⟩ `and` ⟨第二个控制点⟩ `..` ⟨结束点⟩。你可以略掉 `and` ⟨第二个控制点⟩，此时第二个控制点和第一个相同。

所以，Karl 现在可以把一个半圆加进图片了：

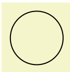


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

Karl 看到结果很开心，但是觉得用这种方法指定圆太笨拙了，不过还好有个简单得多的方法。

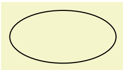
1.5 圆形路径创建

路径创建操作 `circle` 可以用来画圆，后面跟着半径，写在方括号里，例子如下：（注意到前面的点作为圆心。）



```
\tikz \draw (0,0) circle [radius=10pt];
```

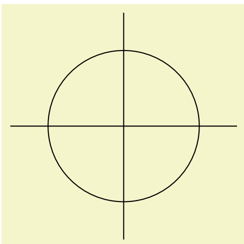
你还可以用 `ellipse` 画椭圆，它有两个半径：



```
\tikz \draw (0,0) ellipse [x radius=20pt, y radius=10pt];
```

要画一个轴线倾斜成任意角度而不是横平竖直的椭圆，比如一个"旋转的椭圆"，你可以加上变换，这个后面会解释。顺带一提，这个小椭圆的代码是：`\tikz \draw[rotate=30] (0,0) ellipse [x radius=6pt, y radius=3pt];`。

所以，回到 Karl 的问题，他可以用 `\draw (0,0) circle [radius=1cm];` 来画圆：

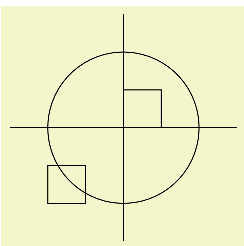


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

此时 Karl 有点担心，因为这个圆太小了，而他最后想要的图片要大得多。他很高兴地了解到，Ti*k*Z 有强大的变换选项，所以把每样东西都放大三倍非常容易。不过我们暂时先保留这个尺寸，省点地方。

1.6 矩形路径创建

我们下一步想画背景中的网格，有好几种方法可以实现。比如你可以画许多矩形。因为矩形很常用，所以有个专门的语法：可以用 `rectangle` 在当前路径中画一个矩形。这个操作后面需要跟另外一个坐标，这样前后两个坐标就构成了矩形对角的两个点，并将矩形绘制出来。所以让我们在图片中加两个矩形：



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (0,0) rectangle (0.5,0.5);
  \draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

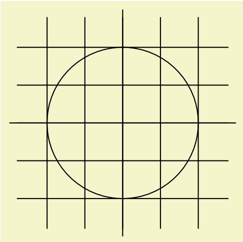
也许在其他情况下这个方法还不错，但是并没有真的解决 Karl 的问题：首先我们需要很多很多矩形，并且边界处是不"闭合"的。

所以，正当 Karl 准备用 `\draw` 命令画四条横线、四条竖线时，他得知有一个 `grid` （网格）路径创建操作。

1.7 网格路径创建

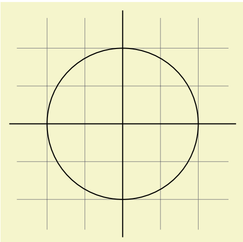
`grid` 路径操作在当前路径加上网格。它会画出组成网格的线，`grid` 前后两个点坐标构成了网格矩形的两个对角。比如，代码 `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` 生成网格 。注意到 `\draw` 后面的选项，可以指定网格的宽度（还可以用 `xstep` 和 `ystep` 来分别设置网格间隔）。正如 Karl 很快会学到的，用这些选项可以影响非常多的事物。

Karl 可以用下面的代码：



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Karl 又看了一眼理想中的图片，他注意到，要是网格更柔和一点就好了。（他儿子告诉他，如果网格不柔和，会让人分心。）为了让网格柔和，他又在 `\draw` 后面加了两个选项。首先他把网格线的颜色设为 `gray`，然后将网格的线宽减到了 `very thin`，最后，他调换了命令的顺序，先画网格，再在上面画其他的。



```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

1.8 加一点样式

```
help lines/.style={color=blue!50,very thin}
```

The effect of this “style setter” is that in the current scope or environment the `help lines` option has the same effect as `color=blue!50,very thin`.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner. Normally, styles are defined at the beginning of a picture. However, you may sometimes wish to define a style globally, so that all pictures of your document can use this style. Then you can easily change the way all graphics look by changing this one style. In this situation you can use the `\tikzset` command at the beginning of the document as in

```
\tikzset{help lines/.style=very thin}
```

To build a hierarchy of styles you can have one style use another. So in order to define a style `Karl's grid` that is based on the `grid` style Karl could say

```
\tikzset{Karl's grid/.style={help lines,color=blue!50}}
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Styles are made even more powerful by parametrization. This means that, like other options, styles can also be used with a parameter. For instance, Karl could parameterize his grid so that, by default, it is blue, but he could also use another color.

```
\begin{tikzpicture}
  [Karl's grid/.style  ={help lines,color=#1!50},
   Karl's grid/.default=blue]

  \draw[Karl's grid]      (0,0) grid (1.5,2);
  \draw[Karl's grid=red]  (2,0) grid (3.5,2);
\end{tikzpicture}
```

1.9 Drawing Options

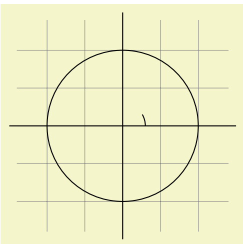
Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

He saw that the style `very thin` yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that `thin` is the correct choice, since it gives the same thickness as TEX's `\hrule` command. Nevertheless, Karl would like to know whether there is anything "in the middle" between `thin` and `thick`. There is: `semithick`.

Another useful thing one can do with lines is to dash or dot them. For this, the two styles `dashed` and `dotted` can be used, yielding - - - - and ········. Both options also exist in a loose and a dense version, called `loosely dashed`, `densely dashed`, `loosely dotted`, and `densely dotted`. If he really, really needs to, Karl can also define much more complex dashing patterns with the `dash pattern` option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

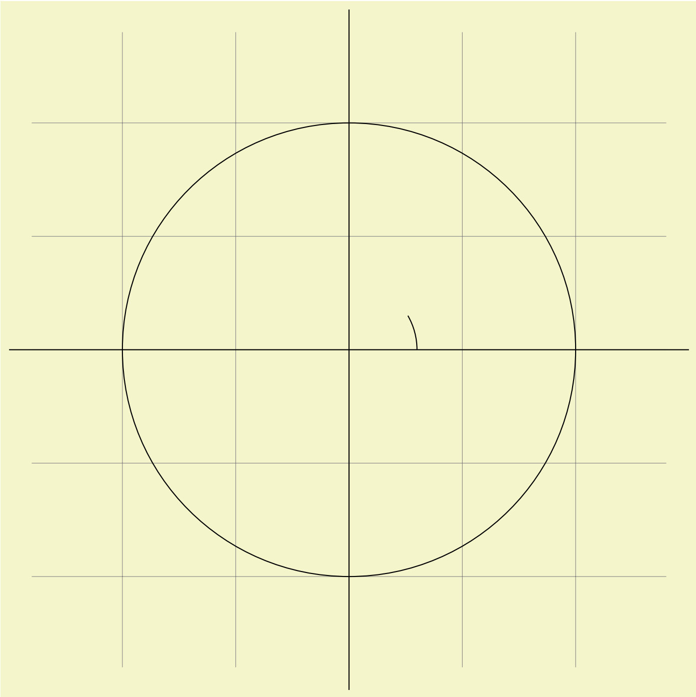
1.10 Arc Path Construction

Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction operation is useful, which draws part of a circle or ellipse. This `arc` operation is followed by options in brackets that specify the arc. An example would be `arc[start angle=10, end angle=80, radius=10pt]`, which means exactly what it says. Karl obviously needs an arc from 0° to 30°. The radius should be something relatively small, perhaps around one third of the circle's radius. When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to "get there."



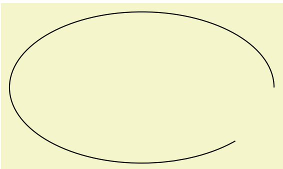
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can add the `[scale=3]` option. He could add this option to each `\draw` command, but that would be awkward. Instead, he adds it to the whole environment, which causes this option to apply to everything within.



```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

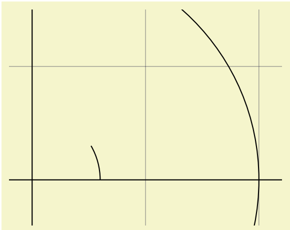
As for circles, you can specify "two" radii in order to get an elliptical arc.



```
\tikz \draw (0,0)
  arc [start angle=0, end angle=315,
       x radius=1.75cm, y radius=1cm];
```

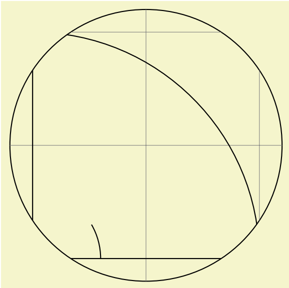
1.11 Clipping a Path

In order to save space in this manual, it would be nice to clip Karl's graphics a bit so that we can focus on the "interesting" parts. Clipping is pretty easy in Ti*k*Z. You can use the `\clip` command to clip all subsequent drawing. It works like `\draw`, only it does not draw anything, but uses the given path to clip everything subsequently.



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

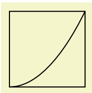
You can also do both at the same time: Draw *and* clip a path. For this, use the `\draw` command and add the `clip` option. (This is not the whole picture: You can also use the `\clip` command and add the `draw` option. Well, that is also not the whole picture: In reality, `\draw` is just a shorthand for `\path[draw]` and `\clip` is a shorthand for `\path[clip]` and you could also say `\path[draw,clip]`.) Here is an example:



```
\begin{tikzpicture}[scale=3]
  \clip[draw] (0.5,0.5) circle (.6cm);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

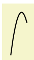
1.12 Parabola and Sine Path Construction

Although Karl does not need them for his picture, he is pleased to learn that there are `parabola` and `sin` and `cos` path operations for adding parabolas and sine and cosine curves to the current path. For the `parabola` operation, the current point will lie on the parabola as well as the point given after the parabola operation. Consider the following example:



```
\tikz \draw (0,0) rectangle (1,1)  (0,0) parabola (1,1);
```

It is also possible to place the bend somewhere else:

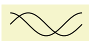


```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

The operations `sin` and `cos` add a sine or cosine curve in the interval $[0, \pi/2]$ such that the previous current point is at the start of the curve and the curve ends at the given end point. Here are two examples:

A sine curve.

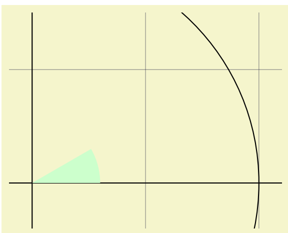
```
A sine \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1); curve.
```



```
\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
                            (0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

1.13 Filling and Drawing

Returning to the picture, Karl now wants the angle to be “filled” with a very light green. For this he uses `\fill` instead of `\draw`. Here is what Karl does:



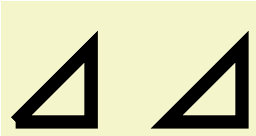
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \fill[green!20!white] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- (0,0);
\end{tikzpicture}
```

The color `green!20!white` means 20% green and 80% white mixed together. Such color expression are possible since Ti*k*Z uses Uwe Kern’s `xcolor` package, see the documentation of that package for details on color expressions.

What would have happened, if Karl had not “closed” the path using `--(0,0)` at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

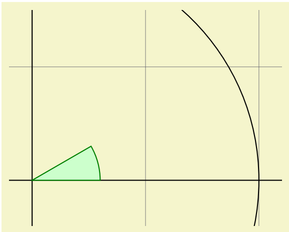
```
\fill[green!20!white] (0,0) -- (3mm,0mm)
  arc [start angle=0, end angle=30, radius=3mm] -- cycle;
```

The `--cycle` causes the current path to be closed (actually the current part of the current path) by smoothly joining the first and last point. To appreciate the difference, consider the following example:



```
\begin{tikzpicture}[line width=5pt]
  \draw (0,0) -- (1,0) -- (1,1) -- (0,0);
  \draw (2,0) -- (3,0) -- (3,1) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

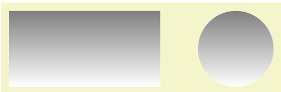
You can also fill and draw a path at the same time using the `\filldraw` command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20!white, draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

1.14 Shading

Karl briefly considers the possibility of making the angle "more fancy" by *shading* it. Instead of filling the area with a uniform color, a smooth transition between different colors is used. For this, `\shade` and `\shadedraw`, for shading and drawing at the same time, can be used:

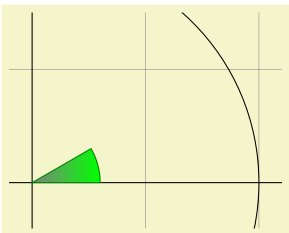


```
  \tikz \shade (0,0) rectangle (2,1)  (3,0.5) circle (.5cm);
```

The default shading is a smooth transition from gray to white. To specify different colors, you can use options:

```
\begin{tikzpicture}[rounded corners,ultra thick]
  \shade[top color=yellow,bottom color=black] (0,0) rectangle +(2,1);
  \shade[left color=yellow,right color=black] (3,0) rectangle +(2,1);
  \shadedraw[inner color=yellow,outer color=black,draw=yellow] (6,0) rectangle +(2,1);
  \shade[ball color=green] (9,.5) circle (.5cm);
\end{tikzpicture}
```

For Karl, the following might be appropriate:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
    (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

However, he wisely decides that shadings usually only distract without adding anything to the picture.

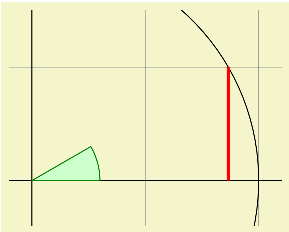
1.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `color=` option to set the lines' colors. So, what is the best way to specify the coordinates?

There are different ways of specifying coordinates. The easiest way is to say something like `(10pt,2cm)`. This means 10pt in x-direction and 2cm in y-directions. Alternatively, you can also leave out the units as in `(1,2)`, which means "one times the current x-vector plus twice the current y-vector." These vectors default to 1cm in the x-direction and 1cm in the y-direction, respectively.

In order to specify points in polar coordinates, use the notation `(30:1cm)`, which means 1cm in direction 30 degree. This is obviously quite useful to "get to the point $(\cos 30°, \sin 30°)$ on the circle."

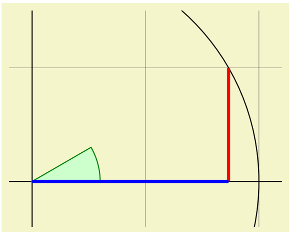
You can add a single `+` sign in front of a coordinate or two of them as in `+(0cm,1cm)` or `++(2cm,0cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making this the new specified position." For example, we can draw the sine line as follows:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick] (30:1cm) -- +(0,-0.5);
\end{tikzpicture}
```

Karl used the fact $\sin 30° = 1/2$. However, he very much doubts that his students know this, so it would be nice to have a way of specifying "the point straight down from `(30:1cm)` that lies on the x-axis." This is, indeed, possible using a special syntax: Karl can write `(30:1cm |- 0,0)`. In general, the meaning of `(`$\langle p \rangle$ `|-` $\langle q \rangle$`)` is "the intersection of a vertical line through p and a horizontal line through q."

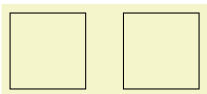
Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we "continue" from where the sine ends:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick]  (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

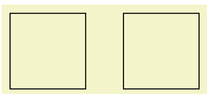
Note that there is no `--` between `(30:1cm)` and `++(0,-0.5)`. In detail, this path is interpreted as follows: "First, the `(30:1cm)` tells me to move by pen to $(\cos 30°, 1/2)$. Next, there comes another coordinate specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30°, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the `--`)."

To appreciate the difference between `+` and `++` consider the following example:



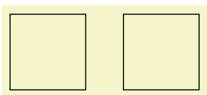
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

By comparison, when using a single `+`, the coordinates are different:



```
\begin{tikzpicture}
  \def\rectanglepath{-- +(1cm,0cm)  -- +(1cm,1cm)  -- +(0cm,1cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Naturally, all of this could have been written more clearly and more economically like this (either with a single of a double `+`):



```
\tikz \draw (0,0) rectangle +(1,1)  (1.5,0) rectangle +(1,1);
```

1.16 Intersecting Paths

Karl is left with the line for $\tan\alpha$, which seems difficult to specify using transformations and polar coordinates. The first – and easiest – thing he can do is so simply use the coordinate `(1,{tan(30)})` since TikZ's math engine knows how to compute things like `tan(30)`. Note the added braces since, otherwise, TikZ's parser would think that the first closing parenthesis ends the coordinate (in general, you need to add braces around components of coordinates when these components contain parentheses).

Karl can, however, also use a more elaborate, but also more "geometric" way of computing the length of the orange line: He can specify intersections of paths as coordinates. The line for $\tan\alpha$ starts at $(1,0)$ and goes upward to a point that is at the intersection of a line going "up" and a line going from the origin through `(30:1cm)`. Such computations are made available by the `intersections` library.

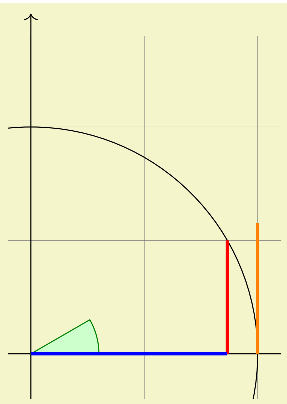
What Karl must do is to create two "invisible" paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the `\path` command without any options like `draw` or `fill`. Then, Karl can add the `name path` option to the path for later reference. Once the paths have been constructed, Karl can use the `name intersections` to assign names to the coordinate for later reference.

```
\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm); % a bit longer, so that there is an intersection

\draw [name intersections={of=upward line and sloped line, by=x}]
  [very thick,orange] (1,0) -- (x);
```

1.17 Adding Arrow Tips

Karl now wants to add the little arrow tips at the end of the axes. He has noticed that in many plots, even in scientific journals, these arrow tips seem to be missing, presumably because the generating programs cannot produce them. Karl thinks arrow tips belong at the end of axes. His son agrees. His students do not care about arrow tips.

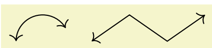
It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
        arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick]    (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick]   (30:1cm) ++(0,-0.5) -- (0,0);

  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=x}]
        [very thick,orange] (1,0) -- (x);
\end{tikzpicture}
```

If Karl had used the option `<-` instead of `->`, arrow tips would have been put at the beginning of the path. The option `<->` puts arrow tips at both ends of the path.

There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open "line." For example, you cannot add tips to, say, a rectangle or a circle. However, you can add arrow tips to curved paths and to paths that have several segments, as in the following examples:



```
\begin{tikzpicture}
  \draw [<->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
  \draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl has a more detailed look at the arrow that TikZ puts at the end. It looks like this when he zooms it: $\rightarrow$. The shape seems vaguely familiar and, indeed, this is exactly the end of TeX's standard arrow used in something like $f\colon A \to B$.

Karl likes the arrow, especially since it is not "as thick" as the arrows offered by many other packages. However, he expects that, sometimes, he might need to use some other kinds of arrow. To do so, Karl can say `>=`⟨*kind of end arrow tip*⟩, where ⟨*kind of end arrow tip*⟩ is a special arrow tip specification. For example, if Karl says `>=Stealth`, then he tells Ti*k*Z that he would like "stealth-fighter-like" arrow tips:

```
\begin{tikzpicture}[>=Stealth]
  \draw [->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
  \draw [<<-,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl wonders whether such a military name for the arrow type is really necessary. He is not really mollified when his son tells him that Microsoft's PowerPoint uses the same name. He decides to have his students discuss this at some point.

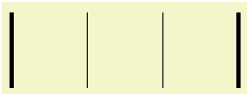
In addition to `Stealth` there are several other predefined kinds of arrow tips Karl can choose from, see Section **??**. Furthermore, he can define arrows types himself, if he needs new ones.

1.18 Scoping

Karl saw already that there are numerous graphic options that affect how paths are rendered. Often, he would like to apply certain options to a whole set of graphic commands. For example, Karl might wish to draw three paths using a `thick` pen, but would like everything else to be drawn "normally."

If Karl wishes to set a certain graphic option for the whole picture, he can simply pass this option to the `\tikz` command or to the `{tikzpicture}` environment (Gerda would pass the options to `\tikzpicture` and Hans passes them to `\starttikzpicture`). However, if Karl wants to apply graphic options to a local group, he put these commands inside a `{scope}` environment (Gerda uses `\scope` and `\endscope`, Hans uses `\startscope` and `\stopscope`). This environment takes graphic options as an optional argument and these options apply to everything inside the scope, but not to anything outside.

Here is an example:



```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (0,1);
  \begin{scope}[thin]
    \draw (1,0) -- (1,1);
    \draw (2,0) -- (2,1);
  \end{scope}
  \draw (3,0) -- (3,1);
\end{tikzpicture}
```

Scoping has another interesting effect: Any changes to the clipping area are local to the scope. Thus, if you say `\clip` somewhere inside a scope, the effect of the `\clip` command ends at the end of the scope. This is useful since there is no other way of "enlarging" the clipping area.

Karl has also already seen that giving options to commands like `\draw` apply only to that command. It turns out that the situation is slightly more complex. First, options to a command like `\draw` are not really options to the command, but they are "path options" and can be given anywhere on the path. So, instead of `\draw[thin] (0,0) --(1,0);` one can also write `\draw (0,0) [thin] --(1,0);` or `\draw (0,0) --(1,0) [thin];`; all of these have the same effect. This might seem strange since in the last case, it would appear that the `thin` should take effect only "after" the line from $(0,0)$ to $(1,0)$ has been drawn. However, most graphic options only apply to the whole path. Indeed, if you say both `thin` and `thick` on the same path, the last option given will "win."

When reading the above, Karl notices that only "most" graphic options apply to the whole path. Indeed, all transformation options do *not* apply to the whole path, but only to "everything following them on the path." We will have a more detailed look at this in a moment. Nevertheless, all options given during a path construction apply only to this path.

1.19 Transformations

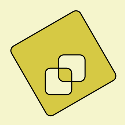
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, Ti*k*Z, TeX, and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the final position on the page.

Ti*k*Z provides numerous options that allow you to transform coordinates in Ti*k*Z's private coordinate system. For example, the `xshift` option allows you to shift all subsequent points by a certain amount:


```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

It is important to note that you can change transformation “in the middle of a path,” a feature that is not supported by PDF or PostScript. The reason is that Ti*k*Z keeps track of its own transformation matrix.

Here is a more complicated example:



```
\begin{tikzpicture}[even odd rule,rounded corners=2pt,x=10pt,y=10pt]
  \filldraw[fill=yellow!80!black] (0,0)   rectangle (1,1)
        [xshift=5pt,yshift=5pt]   (0,0)   rectangle (1,1)
                    [rotate=30]   (-1,-1) rectangle (2,2);
\end{tikzpicture}
```

The most useful transformations are `xshift` and `yshift` for shifting, `shift` for shifting to a given point as in `shift={(1,0)}` or `shift={+(0,0)}` (the braces are necessary so that TeX does not mistake the comma for separating options), `rotate` for rotating by a certain angle (there is also a `rotate around` for rotating around a given point), `scale` for scaling by a certain factor, `xscale` and `yscale` for scaling only in the x- or y-direction (`xscale=-1` is a flip), and `xslant` and `yslant` for slanting. If these transformation and those that I have not mentioned are not sufficient, the `cm` option allows you to apply an arbitrary transformation matrix. Karl's students, by the way, do not know what a transformation matrix is.

1.20 Repeating Things: For-Loops

Karl's next aim is to add little ticks on the axes at positions -1, $-1/2$, $1/2$, and 1. For this, it would be nice to use some kind of “loop,” especially since he wishes to do the same thing at each of these positions. There are different packages for doing this. LaTeX has its own internal command for this, `pstricks` comes along with the powerful `\multido` command. All of these can be used together with Ti*k*Z, so if you are familiar with them, feel free to use them. Ti*k*Z introduces yet another command, called `\foreach`, which I introduced since I could never remember the syntax of the other packages. `\foreach` is defined in the package `pgffor` and can be used independently Ti*k*Z, but Ti*k*Z includes it automatically.

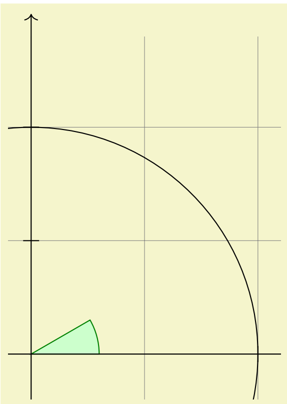
In its basic form, the `\foreach` command is easy to use:

$x = 1$, $x = 2$, $x = 3$,

```
\foreach \x in {1,2,3} {$x =\x$, }
```

The general syntax is `\foreach` ⟨*variable*⟩ `in {`⟨*list of values*⟩`}` ⟨*commands*⟩. Inside the ⟨*commands*⟩, the ⟨*variable*⟩ will be assigned to the different values. If the ⟨*commands*⟩ do not start with a brace, everything up to the next semicolon is used as ⟨*commands*⟩.

For Karl and the ticks on the axes, he could use the following code:

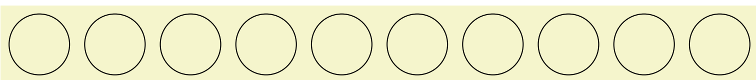


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x in {-1cm,-0.5cm,1cm}
    \draw (\x,-1pt) -- (\x,1pt);
  \foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
    \draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

As a matter of fact, there are many different ways of creating the ticks. For example, Karl could have put the `\draw ...;` inside curly braces. He could also have used, say,

```
\foreach \x in {-1,-0.5,1}
  \draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl is curious what would happen in a more complicated situation where there are, say, 20 ticks. It seems bothersome to explicitly mention all these numbers in the set for `\foreach`. Indeed, it is possible to use `...` inside the `\foreach` statement to iterate over a large number of values (which must, however, be dimensionless real numbers) as in the following example:

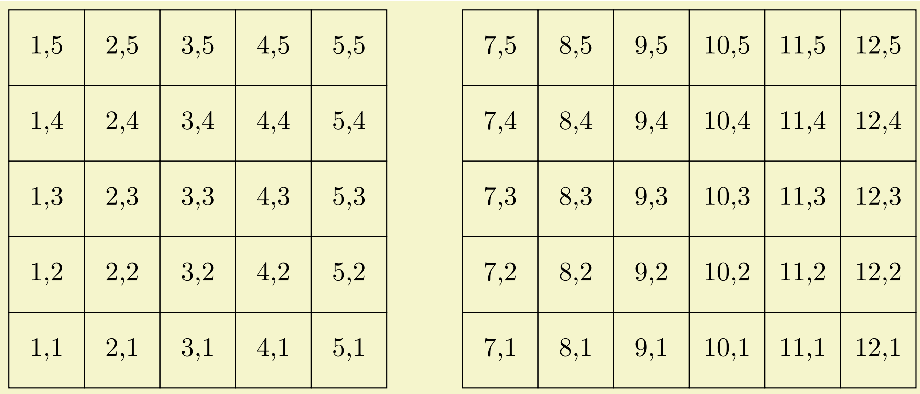


```
\tikz \foreach \x in {1,...,10}
        \draw (\x,0) circle (0.4cm);
```

If you provide *two* numbers before the `...`, the `\foreach` statement will use their difference for the stepping:


```
\tikz \foreach \x in {-1,-0.5,...,1}
      \draw (\x cm,-1pt) -- (\x cm,1pt);
```

We can also nest loops to create interesting effects:



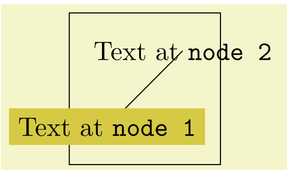

```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

The `\foreach` statement can do even trickier stuff, but the above gives the idea.

1.21 Adding Text

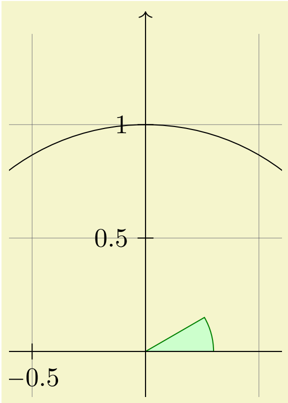
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

Ti*k*Z offers an easy-to-use and powerful system for adding text and, more generally, complex shapes to a picture at specific positions. The basic idea is the following: When Ti*k*Z is constructing a path and encounters the keyword `node` in the middle of a path, it reads a *node specification*. The keyword `node` is typically followed by some options and then some text between curly braces. This text is put inside a normal TEX box (if the node specification directly follows a coordinate, which is usually the case, Ti*k*Z is able to perform some magic so that it is even possible to use verbatim text inside the boxes) and then placed at the current position, that is, at the last specified position (possibly shifted a bit, according to the given options). However, all nodes are drawn only after the path has been completely drawn/filled/shaded/clipped/whatever.




```
\begin{tikzpicture}
  \draw (0,0) rectangle (2,2);
  \draw (0.5,0.5) node [fill=yellow!80!black]
                       {Text at \verb!node 1!}
     -- (1.5,1.5) node {Text at \verb!node 2!};
\end{tikzpicture}
```

Obviously, Karl would not only like to place nodes *on* the last specified position, but also to the left or the right of these positions. For this, every node object that you put in your picture is equipped with several *anchors*. For example, the `north` anchor is in the middle at the upper end of the shape, the `south` anchor is at the bottom and the `north east` anchor is in the upper right corner. When you give the option `anchor=`*north*, the text will be placed such that this northern anchor will lie on the current position and the text is, thus, below the current position. Karl uses this to draw the ticks as follows:

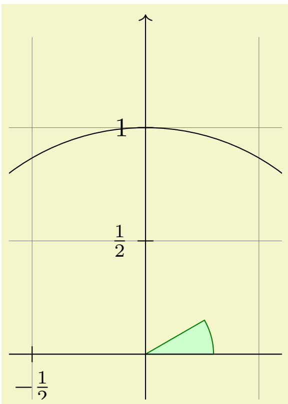



```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x in {-1,-0.5,1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\x$};
  \foreach \y in {-1,-0.5,0.5,1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\y$};
\end{tikzpicture}
```

This is quite nice, already. Using these anchors, Karl can now add most of the other text elements. However, Karl thinks that, though "correct," it is quite counter-intuitive that in order to place something *below* a given point, he has to use the *north* anchor. For this reason, there is an option called `below`, which does the same as `anchor=north`. Similarly, `above right` does the same as `anchor=south west`. In addition, `below` takes an optional dimension argument. If given, the shape will additionally be shifted downwards by the given amount. So, `below=1pt` can be used to put a text label below some point and, additionally shift it 1pt downwards.

Karl is not quite satisfied with the ticks. He would like to have 1/2 or $\frac{1}{2}$ shown instead of 0.5, partly to show off the nice capabilities of TeX and Ti*k*Z, partly because for positions like 1/3 or π it is certainly very much preferable to have the "mathematical" tick there instead of just the "numeric" tick. His students, on the other hand, prefer 0.5 over 1/2 since they are not too fond of fractions in general.

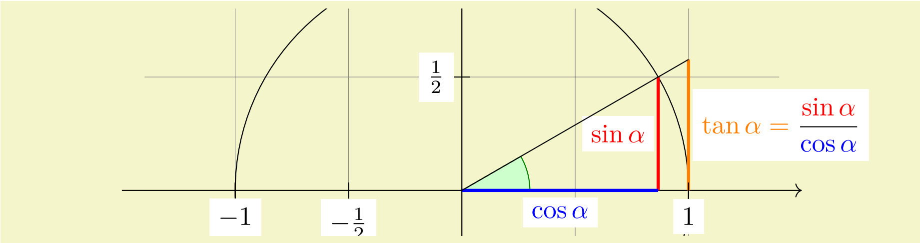
Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead of having one variable `\x`, Karl can specify two (or even more) variables separated by a slash as in `\x / \xtext`. Then, the elements in the set over which `\foreach` iterates must also be of the form ⟨*first*⟩`/`⟨*second*⟩. In each iteration, `\x` will be set to ⟨*first*⟩ and `\xtext` will be set to ⟨*second*⟩. If no ⟨*second*⟩ is given, the ⟨*first*⟩ will be used again. So, here is the new code for the ticks:




```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\ytext$};
\end{tikzpicture}
```

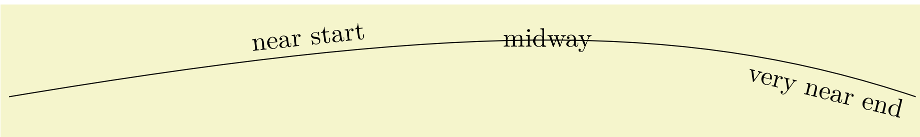
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy solution: Karl can add the `[fill=white]` option to fill out the background of the text shape with a white color.

The next thing Karl wants to do is to add the labels like $\sin\alpha$. For this, he would like to place a label "in the middle of the line." To do so, instead of specifying the label `node {$\sin\alpha$}` directly after one of the endpoints of the line (which would place the label at that endpoint), Karl can give the label directly after the `--`, before the coordinate. By default, this places the label in the middle of the line, but the `pos=` options can be used to modify this. Also, options like `near start` and `near end` can be used to modify this position:




```
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
  \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
  \draw (0,0) circle [radius=1cm];

  \draw[very thick,red]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);
  \draw[very thick,blue]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=t}]
    [very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);

  \draw (0,0) -- (t);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {$\ytext$};
\end{tikzpicture}
```

You can also position labels on curves and, by adding the `sloped` option, have them rotated such that they match the line's slope. Here is an example:




```
\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) ..
    node[near start,sloped,above] {near start}
    node {midway}
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}
```

It remains to draw the explanatory text at the right of the picture. The main difficulty here lies in limiting the width of the text "label," which is quite long, so that line breaking is used. Fortunately, Karl can use the option `text width=6cm` to get the desired effect. So, here is the full code:

```
\begin{tikzpicture}
  [scale=3,line cap=round,
  % Styles
  axes/.style=,
  important line/.style={very thick},
  information text/.style={rounded corners,fill=red!10,inner sep=1ex}]

  % Colors
  \colorlet{anglecolor}{green!50!black}
  \colorlet{sincolor}{red}
  \colorlet{tancolor}{orange!80!black}
  \colorlet{coscolor}{blue}

  % The graphic
  \draw[help lines,step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);

  \draw (0,0) circle [radius=1cm];

  \begin{scope}[axes]
    \draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$} coordinate(x axis);
    \draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);

    \foreach \x/\xtext in {-1, -.5/-\frac{1}{2}, 1}
      \draw[xshift=\x cm] (0pt,1pt) -- (0pt,-1pt) node[below,fill=white] {$\xtext$};

    \foreach \y/\ytext in {-1, -.5/-\frac{1}{2}, .5/\frac{1}{2}, 1}
      \draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
  \end{scope}

  \filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt)
    arc [start angle=0, end angle=30, radius=3mm];
  \draw (15:2mm) node[anglecolor] {$\alpha$};

  \draw[important line,sincolor]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);

  \draw[important line,coscolor]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);

  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=t}]
    [very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);

  \draw (0,0) -- (t);

  \draw[xshift=1.85cm]
    node[right,text width=6cm,information text]
    {
      The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
      example ($\pi/6$ in radians). The {\color{sincolor}sine of
        $\alpha$}, which is the height of the red line, is
      \[
      {\color{sincolor} \sin \alpha} = 1/2.
      \]
      By the Theorem of Pythagoras ...
    };
\end{tikzpicture}
```

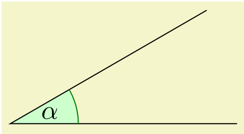
1.22 Pics: The Angle Revisited

Karl expects that the code of certain parts of the picture he created might be so useful that he might wish to reuse them in the future. A natural thing to do is to create TEX macros that store the code he wishes to reuse. However, TikZ offers another way that is integrated directly into its parser: pics!

A “pic” is “not quite a full picture,” hence the short name. The idea is that a pic is simply some code that you can add to a picture at different places using the `pic` command whose syntax is almost identical to the `node` command. The main difference is that instead of specifying some text in curly braces that should be shown, you specify the name of a predefined picture that should be shown.

Defining new pics is easy enough, see Section **??**, but right now we just want to use one such predefined pic: the `angle` pic. As the name suggests, it is a small drawing of an angle consisting of a little wedge and an arc together with some text (Karl needs to load the `angle` library and the `quotes` for the following examples). What makes this pic useful is the fact that the size of the wedge will be computed automatically.

The `angle` pic draws an angle between the two lines BA and BC, where A, B, and C are three coordinates. In our case, B is the origin, A is somewhere on the x-axis and C is somewhere on a line at 30°.




```
\begin{tikzpicture}[scale=3]
  \coordinate (A) at (1,0);
  \coordinate (B) at (0,0);
  \coordinate (C) at (30:1cm);

  \draw (A) -- (B) -- (C)
        pic [draw=green!50!black, fill=green!20, angle radius=9mm,
             "$\alpha$"] {angle = A--B--C};
\end{tikzpicture}
```

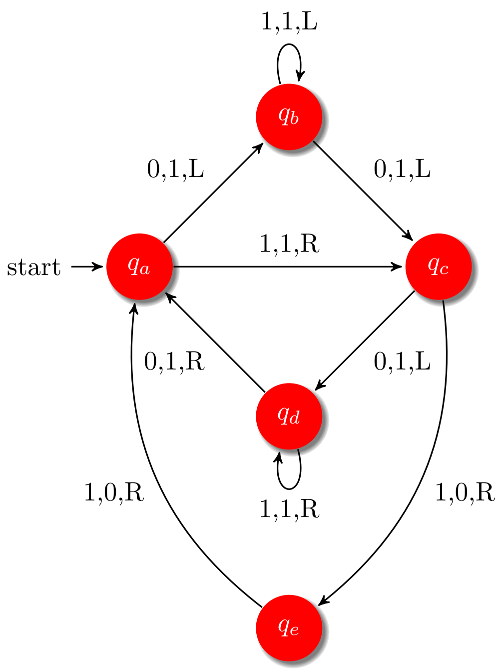
Let us see, what is happening here. First we have specified three *coordinates* using the `\coordinate` command. It allows us to name a specific coordinate in the picture. Then comes something that starts as a normal `\draw`, but then comes the `pic` command. This command gets lots of options and, in curly braces, comes the most important point: We specify that we want to add an `angle` pic and this angle should be between the points we named `A`, `B`, and `C` (we could use other names). Note that the text that we want to be shown in the pic is specified in quotes inside the options of the `pic`, not inside the curly braces.

To learn more about pics, please see Section **??**.

Part II
安装和配置

by Till Tantau

这部分介绍如何安装该系统。通常已经有人帮你装好了，所以你可以跳过这部分；但是如果事与愿违，你是那个不得不自己安装的可怜的家伙，那么请阅读这一部分。



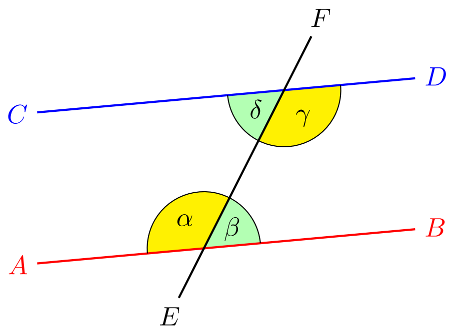
The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet $\{0,1\}$. Proving this conjecture is an open research problem. 中文测试

```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.8cm,on grid,semithick,
                    every state/.style={fill=red,draw=none,circular drop shadow,text=white}]

  \node[initial,state] (A)                    {$q_a$};
  \node[state]         (B) [above right=of A] {$q_b$};
  \node[state]         (D) [below right=of A] {$q_d$};
  \node[state]         (C) [below right=of B] {$q_c$};
  \node[state]         (E) [below=of D]       {$q_e$};

  \path (A) edge              node {0,1,L} (B)
            edge              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) edge [bend left]  node {1,0,R} (A);

   \node [right=1cm,text width=8cm] at (C)
   {
     The current candidate for the busy beaver for five states. It is
     presumed that this Turing machine writes a maximum number of
     $1$'s before halting among all Turing machines with five states
     and the tape alphabet $\{0, 1\}$. Proving this conjecture is an
     open research problem. 中文测试
   };
\end{tikzpicture}
```

Part III

Ti*k*Z ist *kein* Zeichenprogramm

by Till Tantau



When we assume that AB and CD are parallel, i. e., $AB \parallel CD$, then $\alpha = \delta$ and $\beta = \gamma$.

```
\begin{tikzpicture}[angle radius=.75cm]

  \node (A) at (-2,0)    [red,left]    {$A$};
  \node (B) at ( 3,.5)    [red,right]  {$B$};
  \node (C) at (-2,2)  [blue,left]  {$C$};
  \node (D) at ( 3,2.5)  [blue,right] {$D$};
  \node (E) at (60:-5mm) [below]      {$E$};
  \node (F) at (60:3.5cm)  [above]      {$F$};

  \coordinate (X) at (intersection cs:first line={(A)--(B)}, second line={(E)--(F)});
  \coordinate (Y) at (intersection cs:first line={(C)--(D)}, second line={(E)--(F)});

  \path
    (A) edge [red, thick]  (B)
    (C) edge [blue, thick] (D)
    (E) edge [thick]       (F)
      pic ["$\alpha$",  draw, fill=yellow]   {angle = F--X--A}
      pic ["$\beta$",   draw, fill=green!30] {angle = B--X--F}
      pic ["$\gamma$",  draw, fill=yellow]   {angle = E--Y--D}
      pic ["$\delta$",  draw, fill=green!30] {angle = C--Y--E};

  \node at ($ (D)!.5!(B) $) [right=1cm,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
    {
      When we assume that $\color{red}AB$ and $\color{blue}CD$ are
      parallel, i.\,e., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      then $\alpha = \delta$ and $\beta = \gamma$.
    };
\end{tikzpicture}
```

Part IV

Graph Drawing

by Till Tantau et al.

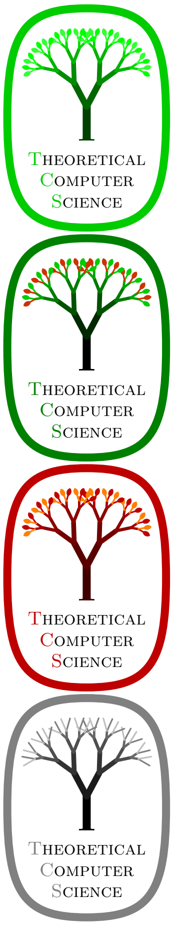
Graph drawing algorithms do the tough work of computing a layout of a graph for you. Ti*k*Z comes with powerful such algorithms, but you can also implement new algorithms in the Lua programming language.

You need to use LuaTEX to typeset this part of the manual (and, also, to use algorithmic graph drawing).

Part V
Libraries

by Till Tantau

In this part the library packages are documented. They provide additional predefined graphic objects like new arrow heads or new plot marks, but sometimes also extensions of the basic PGF or Ti*k*Z system. The libraries are not loaded by default since many users will not need them.

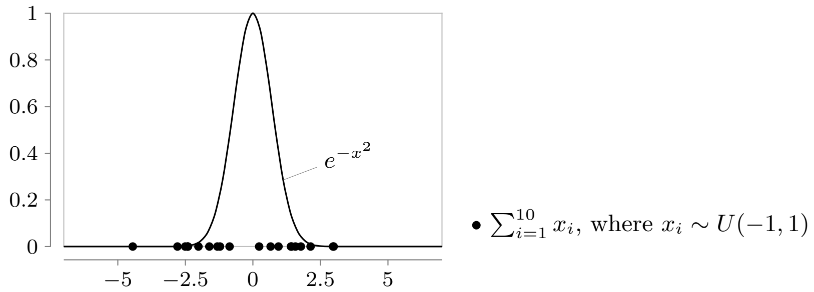



```
\tikzset{
  ld/.style={level distance=#1},lw/.style={line width=#1},
  level 1/.style={ld=4.5mm, trunk,          lw=1ex ,sibling angle=60},
  level 2/.style={ld=3.5mm, trunk!80!leaf a,lw=.8ex,sibling angle=56},
  level 3/.style={ld=2.75mm,trunk!60!leaf a,lw=.6ex,sibling angle=52},
  level 4/.style={ld=2mm,   trunk!40!leaf a,lw=.4ex,sibling angle=48},
  level 5/.style={ld=1mm,   trunk!20!leaf a,lw=.3ex,sibling angle=44},
  level 6/.style={ld=1.75mm,leaf a,         lw=.2ex,sibling angle=40},
}
\pgfarrowsdeclare{leaf}{leaf}
  {\pgfarrowsleftextend{-2pt} \pgfarrowsrightextend{1pt}}
{
  \pgfpathmoveto{\pgfpoint{-2pt}{0pt}}
  \pgfpatharc{150}{30}{1.8pt}
  \pgfpatharc{-30}{-150}{1.8pt}
  \pgfusepathqfill
}

\newcommand{\logo}[5]
{
  \colorlet{border}{#1}
  \colorlet{trunk}{#2}
  \colorlet{leaf a}{#3}
  \colorlet{leaf b}{#4}
  \begin{tikzpicture}
    \scriptsize\scshape
    \draw[border,line width=1ex,yshift=.3cm,
          yscale=1.45,xscale=1.05,looseness=1.42]
      (1,0) to [out=90, in=0]    (0,1)  to [out=180,in=90]  (-1,0)
            to [out=-90,in=-180] (0,-1) to [out=0,  in=-90] (1,0) -- cycle;

    \coordinate (root) [grow cyclic,rotate=90]
    child {
      child [line cap=round] foreach \a in {0,1} {
        child foreach \b in {0,1} {
          child foreach \c in {0,1} {
            child foreach \d in {0,1} {
              child foreach \leafcolor in {leaf a,leaf b}
                { edge from parent [color=\leafcolor,-#5] }
        } } }
      } edge from parent [shorten >=-1pt,serif cm-,line cap=butt]
    };

    \node [align=center,below] at (0pt,-.5ex)
    { \textcolor{border}{T}heoretical \\ \textcolor{border}{C}omputer \\
      \textcolor{border}{S}cience };
  \end{tikzpicture}
}
\begin{minipage}{3cm}
  \logo{green!80!black}{green!25!black}{green}{green!80}{leaf}\\
  \logo{green!50!black}{black}{green!80!black}{red!80!green}{leaf}\\
  \logo{red!75!black}{red!25!black}{red!75!black}{orange}{leaf}\\
  \logo{black!50}{black}{black!50}{black!25}{}
\end{minipage}
```

Part VI

Data Visualization

by Till Tantau

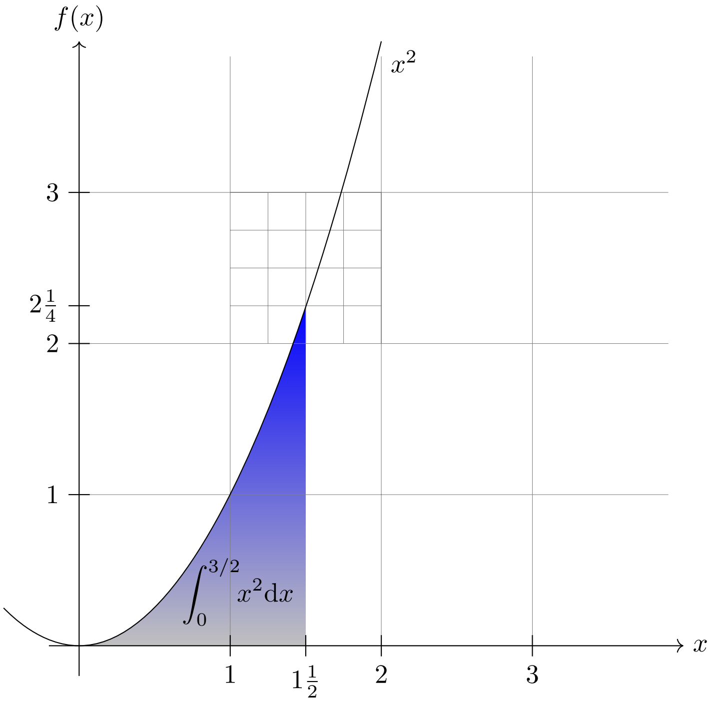



```
\tikz \datavisualization [scientific axes=clean]
[
  visualize as smooth line=Gaussian,
  Gaussian={pin in data={text={$e^{-x^2}$},when=x is 1}}
]
data [format=function] {
  var x : interval [-7:7] samples 51;
  func y = exp(-\value x*\value x);
}
[
  visualize as scatter,
  legend={south east outside},
  scatter={
    style={mark=*,mark size=1.4pt},
    label in legend={text={
        $\sum_{i=1}^{10} x_i$, where $x_i \sim U(-1,1) $}}}
]
data [format=function] {
  var i : interval [0:1] samples 20;
  func y = 0;
  func x = (rand + rand + rand + rand + rand +
            rand + rand + rand + rand + rand);
};
```

Part VII
Utilities

by Till Tantau

The utility packages are not directly involved in creating graphics, but you may find them useful nonetheless. All of them either directly depend on PGF or they are designed to work well together with PGF even though they can be used in a stand-alone way.




```
\begin{tikzpicture}[scale=2]
  \shade[top color=blue,bottom color=gray!50] (0,0) parabola (1.5,2.25) |- (0,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \!\!x^2\mathrm{d}x$};

  \draw[help lines] (0,0) grid (3.9,3.9)
       [step=0.25cm]      (1,2) grid +(1,1);

  \draw[->] (-0.2,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4) node[above] {$f(x)$};

  \foreach \x/\xtext in {1/1, 1.5/1\frac{1}{2}, 2/2, 3/3}
    \draw[shift={(\x,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\xtext$};

  \foreach \y/\ytext in {1/1, 2/2, 2.25/2\frac{1}{4}, 3/3}
    \draw[shift={(0,\y)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\ytext$};

  \draw (-.5,.25) parabola bend (0,0) (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

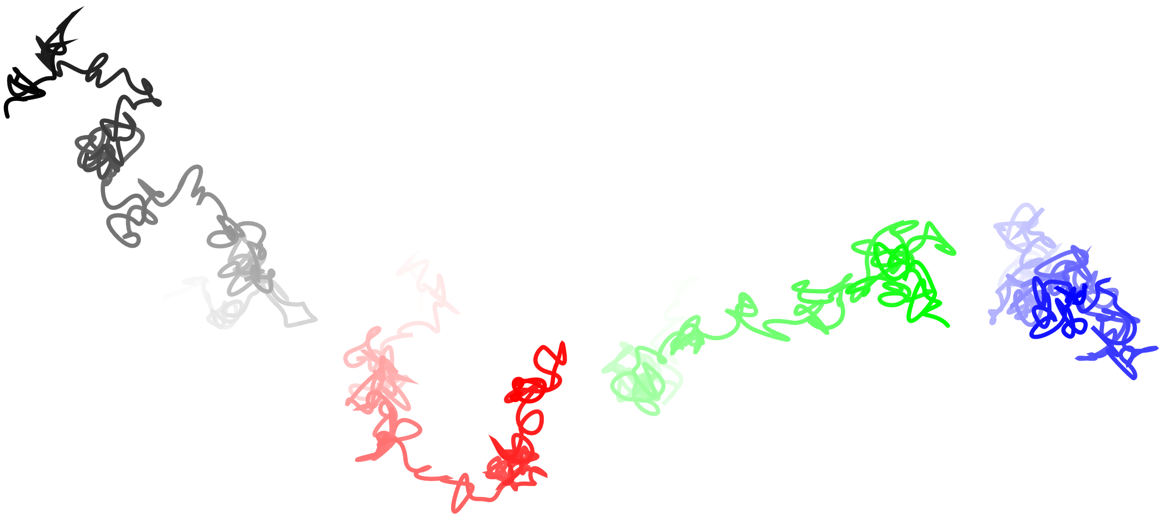
Part VIII
Mathematical and Object-Oriented Engines

by Mark Wibrow and Till Tantau

PGF comes with two useful engines: One for doing mathematics, one for doing object-oriented programming. Both engines can be used independently of the main PGF.

The job of the mathematical engine is to support mathematical operations like addition, subtraction, multiplication and division, using both integers and non-integers, but also functions such as square-roots, sine, cosine, and generate pseudo-random numbers. Mostly, you will use the mathematical facilities of PGF indirectly, namely when you write a coordinate like `(5cm*3,6cm/4)`, but the mathematical engine can also be used independently of PGF and Ti*k*Z.

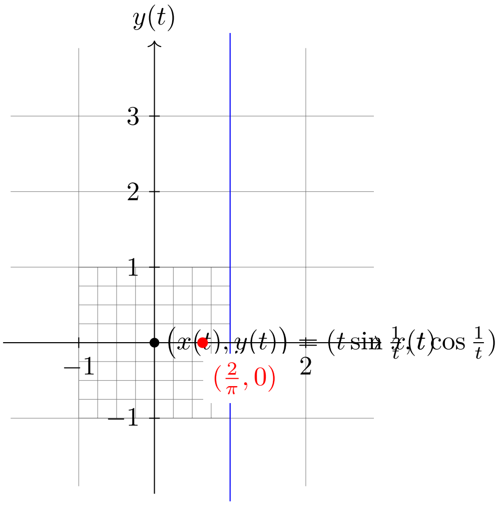
The job of the object-oriented engine is to support simple object-oriented programming in TeX. It allows the definition of *classes* (without inheritance), *methods*, *attributes* and *objects*.



```
\pgfmathsetseed{1}
\foreach \col in {black,red,green,blue}
{
  \begin{tikzpicture}[x=10pt,y=10pt,ultra thick,baseline,line cap=round]
    \coordinate (current point) at (0,0);
    \coordinate (old velocity) at (0,0);
    \coordinate (new velocity) at (rand,rand);

    \foreach \i in {0,1,...,100}
    {
      \draw[\col!\i] (current point)
      .. controls ++([scale=-1]old velocity) and
                  ++(new velocity) .. ++(rand,rand)
         coordinate (current point);
      \coordinate (old velocity) at (new velocity);
      \coordinate (new velocity) at (rand,rand);
    }
  \end{tikzpicture}
}
```

Part IX
The Basic Layer

by Till Tantau




```
\begin{tikzpicture}
  \draw[gray,very thin] (-1.9,-1.9) grid (2.9,3.9)
          [step=0.25cm] (-1,-1) grid (1,1);
  \draw[blue] (1,-2.1) -- (1,4.1); % asymptote

  \draw[->] (-2,0) -- (3,0) node[right] {$x(t)$};
  \draw[->] (0,-2) -- (0,4) node[above] {$y(t)$};

  \foreach \pos in {-1,2}
    \draw[shift={(\pos,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\pos$};

  \foreach \pos in {-1,1,2,3}
    \draw[shift={(0,\pos)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\pos$};

  \fill (0,0) circle (0.064cm);
  \draw[thick,parametric,domain=0.4:1.5,samples=200]
    % The plot is reparameterised such that there are more samples
    % near the center.
    plot[id=asymptotic-example] function{(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))}
    node[right] {$\bigl(x(t),y(t)\bigr) = (t\sin \frac{1}{t}, t\cos \frac{1}{t})$};

  \fill[red] (0.63662,0) circle (2pt)
    node [below right,fill=white,yshift=-4pt] {$(\frac{2}{\pi},0)$};
\end{tikzpicture}
```

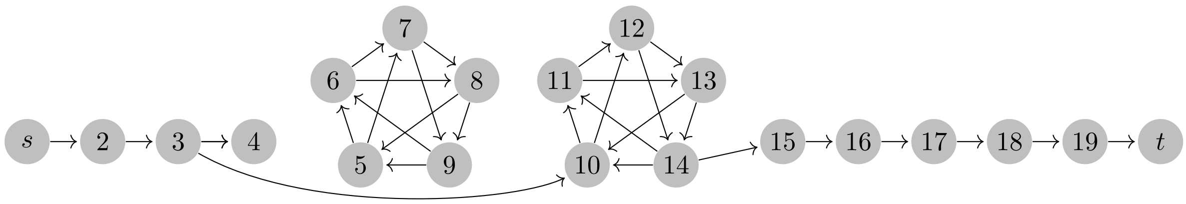
Part X
The System Layer

by Till Tantau

This part describes the low-level interface of PGF, called the *system layer*. This interface provides a complete abstraction of the internals of the underlying drivers.

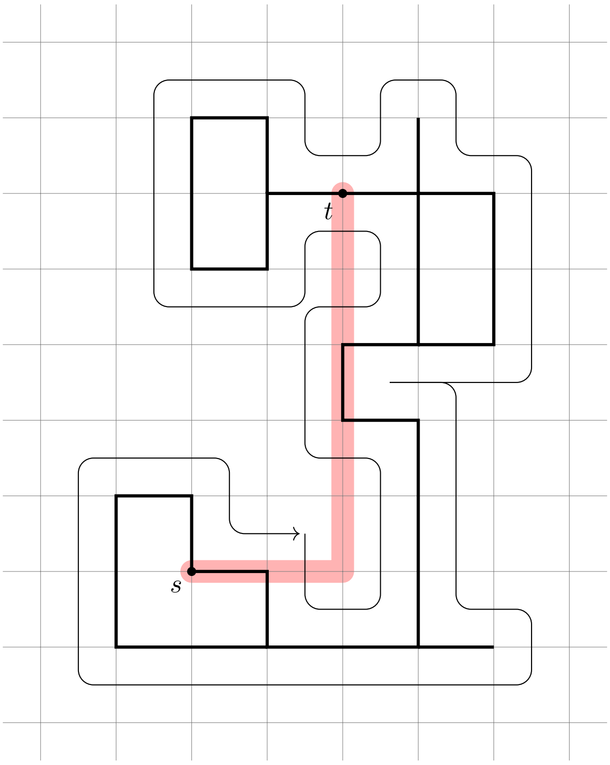
Unless you intend to port PGF to another driver or unless you intend to write your own optimized frontend, you need not read this part.

In the following it is assumed that you are familiar with the basic workings of the `graphics` package and that you know what TeX-drivers are and how they work.




```
\begin{tikzpicture}
  [shorten >=1pt,->,
   vertex/.style={circle,fill=black!25,minimum size=17pt,inner sep=0pt}]

  \foreach \name/\x in {s/1, 2/2, 3/3, 4/4, 15/11, 16/12, 17/13, 18/14, 19/15, t/16}
    \node[vertex] (G-\name) at (\x,0) {$\name$};

  \foreach \name/\angle/\text in {P-1/234/5, P-2/162/6, P-3/90/7, P-4/18/8, P-5/-54/9}
    \node[vertex,xshift=6cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \name/\angle/\text in {Q-1/234/10, Q-2/162/11, Q-3/90/12, Q-4/18/13, Q-5/-54/14}
    \node[vertex,xshift=9cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \from/\to in {s/2,2/3,3/4,3/4,15/16,16/17,17/18,18/19,19/t}
    \draw (G-\from) -- (G-\to);

  \foreach \from/\to in {1/2,2/3,3/4,4/5,5/1,1/3,2/4,3/5,4/1,5/2}
    { \draw (P-\from) -- (P-\to); \draw (Q-\from) -- (Q-\to); }

  \draw (G-3) .. controls +(-30:2cm) and +(-150:1cm) .. (Q-1);
  \draw (Q-5) -- (G-15);
\end{tikzpicture}
```

Part XI
References and Index




```
\begin{tikzpicture}
  \draw[line width=0.3cm,color=red!30,line cap=round,line join=round] (0,0)--(2,0)--(2,5);
  \draw[help lines] (-2.5,-2.5) grid (5.5,7.5);
  \draw[very thick] (1,-1)--(-1,-1)--(-1,1)--(0,1)--(0,0)--
    (1,0)--(1,-1)--(3,-1)--(3,2)--(2,2)--(2,3)--(3,3)--
    (3,5)--(1,5)--(1,4)--(0,4)--(0,6)--(1,6)--(1,5)
    (3,3)--(4,3)--(4,5)--(3,5)--(3,6)
    (3,-1)--(4,-1);
  \draw[below left] (0,0) node(s){$s$};
  \draw[below left] (2,5) node(t){$t$};
  \fill (0,0) circle (0.06cm) (2,5) circle (0.06cm);
  \draw[->,rounded corners=0.2cm,shorten >=2pt]
    (1.5,0.5)-- ++(0,-1)-- ++(1,0)-- ++(0,2)-- ++(-1,0)-- ++(0,2)-- ++(1,0)--
    ++(0,1)-- ++(-1,0)-- ++(0,-1)-- ++(-2,0)-- ++(0,3)-- ++(2,0)-- ++(0,-1)--
    ++(1,0)-- ++(0,1)-- ++(1,0)-- ++(0,-1)-- ++(1,0)-- ++(0,-3)-- ++(-2,0)--
    ++(1,0)-- ++(0,-3)-- ++(1,0)-- ++(0,-1)-- ++(-6,0)-- ++(0,3)-- ++(2,0)--
    ++(0,-1)-- ++(1,0);
\end{tikzpicture}
```