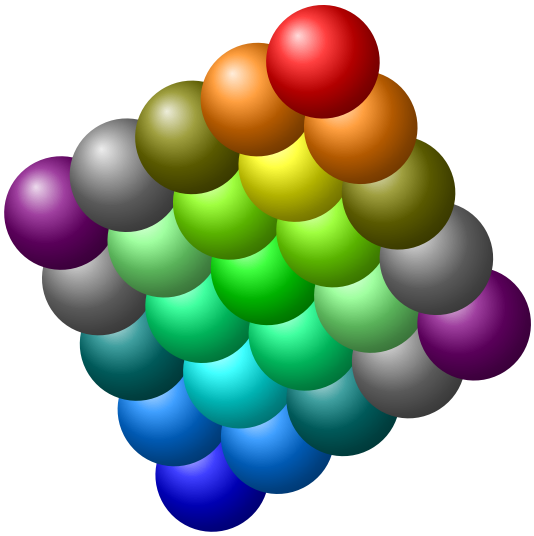
Ti*k*Z and PGF

Manual for version 0.95



```
\tikz[rotate=30]
  \foreach \x / \xcolor in {0/blue,1/cyan,2/green,3/yellow,4/red}
    \foreach \y / \ycolor in {0/blue,1/cyan,2/green,3/yellow,4/red}
      \shade[ball color=\xcolor!50!\ycolor] (\x,\y) circle (7.5mm);
```

Für meinen Vater, damit er noch viele schöne TEX-Graphiken erschaffen kann.

The Ti*k*Z and PGF Packages
Manual for Version 0.95

http://latex-beamer.sourceforge.net

Till Tantau
tantau@users.sourceforge.net

June 12, 2005

Contents

1 Introduction

The PGF package, where "PGF" is supposed to mean "portable graphics format" (or "pretty, good, functional" if you prefer...), is a package for creating graphics in an "inline" manner. The package defines a number of TEX-commands that draw graphics. For example, the code `\tikz \draw (0pt,0pt) -- (20pt,6pt);` yields the line and the code `\tikz \fill[orange] (1ex,1ex) circle (1ex);` yields  .

In a sense, when using PGF, you "program" your graphics, just as you "program" your document when using TEX. This means that you get the advantages of the "TEX-approach to typesetting" also for your graphics: quick creating of simple graphics, precise positioning, the use of macros, often superiour typography. You also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, code does not really "show" how things will look like.

1.1 Structure of the System

The PGF system consists of different layers:

System layer: This layer provides a complete abstraction of what is going on "in the driver." The driver is a program like `dvips` or `dvipdfm` that takes a `.dvi` file as input and generates a `.ps` or a `.pdf` file. (The `pdftex` program also counts as a driver, even though it does not take a `.dvi` file as input. Never mind.) Each driver has its own syntax for the generation of graphics, causing headaches to everyone who wants to create graphics in a portable way. PGF's system layer "abstracts away" these differences. For example, the system command `\pgfsys@lineto{10}{10}` extends the current path to the coordinate (10bp, 10bp) of the current `{pgfpicture}` ("bp" is TEX's "big point" unit). Depending on whether `dvips`, `dvipdfm`, or `pdftex` is used to process the document, the system command will be converted to different `\special` commands.

The system layer is as "minimalistic" as possible since each additional command makes it more work to port PGF to a new driver. Currently, only drivers that produce PostScript or PDF output are supported and only few of them (hence the name *portable* graphics format is currently a bit boastful). However, in principle, the system layer could be ported to many different drivers quite easily. It should even be possible to produce, say, SVG output in conjunction with TEX4HT.

As a user, you will not use the system layer directly.

Basic layer: The basic layer provides a set of basic commands that allow you to produce complex graphics in a much easier way than by using the system layer directly. For example, the system layer provides no commands for creating circles since circles can be composed from the more basic Beziér curves (well, almost). However, as a user you will want to have a simple command to create circles (at least I do) instead of having to write down half a page of Beziér curve support coordinates. Thus, the basic layer provides a command `\pgfpathcircle` that generates the necessary curve coordinates for you.

The basic layer is consists of a *core*, which consists of several interdependent packages that can only be loaded *en bloc* and add-on packages that extend the core by more special-purpose commands like node management or a plotting interface. For example, the BEAMER package uses the core, but not all of the add-one packages of the basic layer.

Frontend layer: A frontend (of which there can be several) is mainly a set of commands or a special syntax that makes using the basic layer easier. A problem with directly using the basic layer is that code written for this layer is often too "verbose." For example, to draw a simple triangle, you may need as many as five commands when using the basic layer: One for beginning a path at the first corner of the triangle, one for extending the path to the second corner, one for going to the third, one for closing the path, and one for actually painting the triangle (as opposed to filling it). With the `tikz` frontend all this boils down to a single, simple METAFONT-like command:

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

There are different frontends:

- The Ti*k*Z frontend is the "natural" frontend for PGF. It gives you access to all features of PGF, but it is intended to be easy to use. The syntax is a mixture of METAFONT and PSTRICKS and some ideas of myself. This frontend is *neither* a complete METAFONT compatiblity layer nor a PSTRICKS compatiblity layer and it is not intended to become either.

- The `pgfpict2e` frontend reimplements the standard LaTeX `{picture}` environment and commands like `\line` or `\vector` using the PGF basic layer. This layer is not really "necessary" since the `pict2e.sty` package does at least as good a job at reimplementing the `{picture}` environment. Rather, the idea behind this package is to have a simple demonstration of how a frontend can be implemented.

It would be possible to implement a `pgftricks` frontend that maps PSTRICKS commands to PGF commands. However, I have not done this and even if fully implemented, many things that work in PSTRICKS will not work, namely whenever some PSTRICKS command relies too heavily on PostScript trickery. Nevertheless, such a package might be useful in some situations.

As a user of PGF you will use the commands of one or several of the frontends plus perhaps some commands of the basic layer. For this reason, this manual explains the frontends first, then the basic layer, and finally the system layer.

1.2 Comparison with Other Graphics Packages

There were two main motivations for creating PGF:

1. The standard LaTeX `{picture}` environment is not powerful enough to create anything but really simple graphics. This is certainly not due to a lack of knowledge or imagination on the part of LaTeX's designer(s). Rather, this is the price paid for the `{picture}` environment's portability: It works together with all backend drivers.

2. The `{pstricks}` package is certainly powerful enough to create any conceivable kind of graphic, but it is not portable at all. Most importantly, it does not work with `pdftex` nor with any other driver that produces anything but PostScript code for that matter.

The PGF package is a tradeoff between portability and expressive power. It is not as portable as `{picture}` and not as powerful as `{pspicture}`. However, it is more powerful than `{picture}` and more portable than `{pspicture}`.

1.3 Supported TeX-Formats

PGF can be used with any TeX-format that is based on Donald Knuth's original `plain` format. This includes LaTeX and ConTeXt. If you use any format other than LaTeX, you must you must say `\input tikz.tex` and `\input pgf.tex` instead of `\usepackage{tikz}` or `\usepackage{pgf}` and you must say `\pgfpicture` instead of `\begin{pgfpicture}` and `\endpicture` instead of `\end{pgfpicture}`.

PGF was originally written for use with LaTeX and this shows in a number of places. Nevertheless, the plain TeX support is reasonably good.

1.4 Utilities: Page Management

The PGF package include a special subpackage called `pgfpages`, which is used to assemble several pages into a single page. This package is not really about creating graphics, but it is part of PGF nevertheless, mostly because its implementation uses PGF heavily.

The subpackage `pgfpages` provides commands for assembiling several "virtual pages" into a single "physical page." The idea is that whenever TeX has a page ready for "shipout," `pgfpages` interrupts this shipout and instead stores the page to be shipped out in a special box. When enough "virtual pages" have been accumulated in this way, they are scaled down and arranged on a "physical page," which then *really* shipped out. This mechanism allows you to create "two page on one page" versions of a document directly inside LaTeX without the use of any external programs.

However, `pgfpages` can do quite a lot more than that. You can use it to put logos and watermark on pages, print up to 16 pages on one page, add borders to pages, and more.

1.5 How to Read This Manual

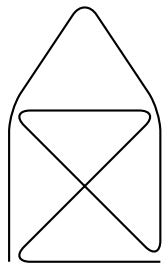
This manual describes both the design of the PGF system and its usage. The organisation is very roughly according to "user-friendliness." The commands and subpackages that are easiest and most frequently used are described first, more low-level and esoteric features are discussed later.

If you have not yet installed PGF, please read the installation first. Second, it might be a good idea to read the tutorial. Finally, you might wish to skim through the description of Ti*k*Z. Typically, you will not need to read the sections on the basic layer. You will only need to read the part on the system layer if you intend to write your own frontend or if you wish to port PGF to a new driver.

The "public" commands and environments provided by the `pgf` package are described throughout the text. In each such description, the described command, environment or option is printed in red. Text shown in green is optional and can be left out.

Part I
Getting Started

This part is intended to help you get started with the PGF package. First, the installation process is explained; however, the system will typically be already installed on your system, so this can often be skipped. Next, a short tutorial is given that explains the most often used commands and concepts of PGF, without going into any of the glorious details.



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.5) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

2 Installation

There are different ways of installing PGF, depending on your system and needs, and you may need to install other packages as well as, see below. Before installing, you may wish to review the GPL license under which the package is distributed, see Section 2.5.

Typically, the package will already be installed on your system. Naturally, in this case you do not need to worry about the installation process at all and you can skip the rest of this section.

2.1 Package and Driver Versions

This documentation is part of version 0.95 of the PGF pacakge. In order to run PGF, you need a reasonably recent version TEX installation. When using LATEX, you need the following packages installed (newer versions should also work):

- `xcolor` version 2.00.
- `xkeyval` version 1.8, if you wish to use Ti*k*Z.

With plain TEX, `xcolor` is not needed, but you obviously do not get its functionality.

Currently, PGF supports the following backend drivers:

- `pdftex` version 0.14 or higher. Earlier versions do not work.
- `dvips` version 5.94a or higher. Earlier versions may also work.
- `dvipdfm` version 0.13.2c or higher. Earlier versions may also work.

Support for other drivers, like VTEX, is under construction, but no promises as to whether or when they will be supported.

Currently, PGF supports the following formats:

- `latex` with complete functionality.
- `plain` with complete functionality, except for graphics inclusion, which works only for pdfTEX.
- `context` should work as `plain`, but I have not tried it.

2.2 Installing Prebundled Packages

I do not create or manage prebundled packages of PGF, but, fortunately, nice other people do. I cannot give detailed instructions on how to install these packages, since I do not manage them, but I *can* tell you were to find them. If you have a problem with installing, you might wish to have a look at the Debian page or MikTeX page first.

2.2.1 Debian

The command "`aptitude install pgf`" should do the trick. If necessary, the package `latex-xcolor` will be automatically installed. Sit back and relax. In detail, the following packages are installed:

`http://packages.debian.org/pgf`
`http://packages.debian.org/latex-xcolor`

2.2.2 MiKTeX

For MiKTEX, use the update wizard to install the (latest versions of the) packages called `pgf`, and `xcolor`.

2.3 Installation in a texmf Tree

For a more permanent installation, you can place the files of the the PGF package in an appropriate `texmf` tree.

When you ask TEX to use a certain class or package, it usually looks for the necessary files in so-called `texmf` trees. These trees are simply huge directories that contain these files. By default, TEX looks for files in three different `texmf` trees:

- The root `texmf` tree, which is usually located at `/usr/share/texmf/` or `c:\texmf\` or somewhere similar.
- The local `texmf` tree, which is usually located at `/usr/local/share/texmf/` or `c:\localtexmf\` or somewhere similar.
- Your personal `texmf` tree, which is usually located in your home directory at `~/texmf/` or `~/Library/texmf/`.

You should install the packages either in the local tree or in your personal tree, depending on whether you have write access to the local tree. Installation in the root tree can cause problems, since an update of the whole TEX installation will replace this whole tree.

2.3.1 Installation that Keeps Everything Together

Once you have located the right texmf tree, you must decide whether you want to install PGF in such a way that "all its files are kept in one place" or whether you want to be "TDS-compliant," where TDS means "TEX directory structure."

If you want to keep "everything in one place," inside the `texmf` tree you have chosen create a sub-sub-directory called `texmf/tex/generic/pgf` or `texmf/tex/generic/pgf-0.95`, if you prefer. Then place all files of the `pgf` package in this directory. Finally, rebuild TEX's filename database. This done by running the command `texhash` or `mktexlsr` (they are the same). In MikTeX, there is a menu option to do this.

2.3.2 Installation that is TDS-Compliant

While the above installation process is the most "natural" one and although I would like to recommend it since it makes updating and managing the PGF package easy, it is not TDS-compliant. If you want to be TDS-compliant, proceed as follows: (If you do not know what TDS-compliant means, you propably do not want to be TDC-compliant.)

The `.tar` file of the `pgf` package contains the following files and directories at its root: `README`, `doc`, `generic`, `plain`, and `latex`. You should "merge" each of the four directories with the following directories `texmf/doc`, `texmf/tex/generic`, `texmf/tex/plain`, and `texmf/tex/latex`. For example, in the `.tar` file the `doc` directory contains just the directory `pgf`, and this directory has to be moved to `texmf/doc/pgf`. The root `README` file can be ignored since it is already reproduced in `doc/pgf/README`.

For a more detailed explanation of the standard installation process of packages, you might wish to consult `http://www.ctan.org/installationadvice/`. However, note that the PGF package does not come with a `.ins` file (simply skip that part).

2.4 Updating the Installation

To update your installation from a previous version, all you need to do is to replace everything in the directory `texmf/tex/generic/pgf` with the files of the new version (or in all the directories where `pgf` was installed, if you chose a TDS-compliant installation). The easiest way to do this is to first delete the old version and then proceed as described above. Sometimes, there are changes in the syntax of certain command from version to version. If things no longer work that used to work, you may wish to have a look at the release notes and at the change log.

2.5 License: The GNU Public License, Version 2

The PGF package is distributed under the GNU public license, version 2. In detail, this means the following (the following text is copyrighted by the Free Software Foundation):

2.5.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

2.5.2 Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the

Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

2.5.3 No Warranty

10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the

use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

3 Tutorial: A Picture for Karl's Students

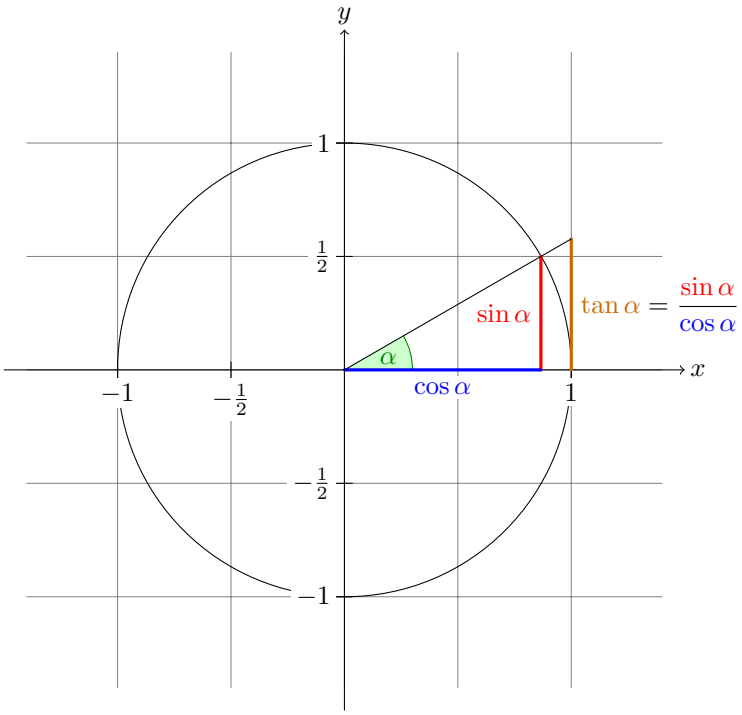
This tutorial is intended for new users of PGF and Ti*k*Z. It does not give an exhaustive account of all the features of Ti*k*Z or PGF, just of those that you are likely to use right away.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using LaTeX's `{picture}` environment. While the results were acceptable, creating the graphics often turned out to be a lengthy process. Also, there tended to be some problems with lines having slightly wrong angles and circles also seemed to be hard to get right. Naturally, his students could not care less whether the lines had the exact right angles and they find Karl's exams too difficult no matter how nicely they were drawn. But Karl was never entirely satisfied with the result.

Karl's son, who was even less satisfied with the results (he did not have to take the exams, after all), told Karl that he might wish to try out a new package for creating graphics. A bit confusingly, this package seemed to have two names: First, Karl had to download and install a package called PGF. Then it turns out that inside this package there is another package called Ti*k*Z, which is supposed to stand for "Ti*k*Z ist *kein* Zeichenprogramm." Karl finds this all a bit strange and Ti*k*Z seems to indicate that the package exactly does not do what he needs. However, having used GNU software for quite some time and "GNU not being Unix," there seems to be hope yet. His son assures him that Ti*k*Z's name is intended only to tell people that Ti*k*Z is not a program that you can use to draw graphics with your mouse or tablet. Rather, it is more like a "graphics language."

3.1 Problem Statement

Karl wants to put a graphic on the next worksheet for his students. He is currently teaching his students about sine and cosine. What he would like to have is something that looks like this (ideally):



The angle α is 30° in the example ($\pi/6$ in radians). The sine of α, which is the height of the red line, is

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

By the Theorem of Phythagoras we have $\cos^2\alpha + \sin^2\alpha = 1$. Thus the length of the blue line, which is the cosine of α, must be

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

This shows that $\tan\alpha$, which is the height of the orange line, is

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{2}.$$

3.2 Setting up the Environment

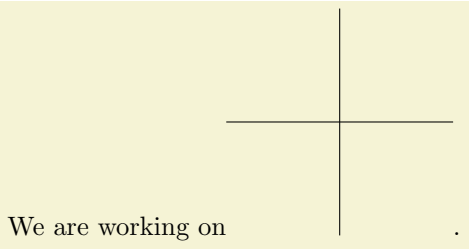
In Ti*k*Z, to draw a picture, at the start of the picture you need to tell TeX or LaTeX that you want to start a picture. In LaTeX this is done using the environment `{tikzpicture}`, in plain TeX you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

3.2.1 Setting up the Environment in LaTeX

Karl, being a LaTeX user, thus sets up his file as follows:

```
\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}
```

When executed, that is, run via `pdflatex` or via `latex` followed by `dvips`, the resulting will contain something that looks like this:




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

Admittedly, not quite the whole picture, yet, but we do have the axes established. Well, not quite, but we have the lines that make up the axes drawn. Karl suddenly has a sinking feeling that the picture is still some way off.

Let's have a more detailed look at the code. First, the package `tikz` is loaded. This package is a so-called "frontend" to the basic PGF system. The basic layer, which is also described in this manual, is somewhat more, well, basic and thus harder to use. The frontend makes things easier by providing a simpler syntax.

Inside the environment, there are two `\draw` commands. They mean: "The path, which is specified following the command up to the semicolon, should be drawn." The first path is specified as `(-1.5,0) -- (0,1.5)`, which means "a straight line from the point at position $(-1.5, 0)$ to a point at position $(0, 1.5)$." Here, the positions are specified within a special coordinate system in which, initially, one unit is 1cm.

Karl is quite pleased to note that the environment automatically reserves enough space to encompass the picture.

3.2.2 Setting up the Environment in Plain TEX

Karl's wife Gerda, who also happens to be a math teacher, is not a LATEX user, but uses plain TEX since she prefers to do things "the old way." She can also use Ti*k*Z. Instead of `\usepackage{tikz}` she has to write `\input tikz.tex` and instead of `\begin{tikzpicture}` she writes `\tikzpicture` and instead of `\end{tikzpicture}` she writes `\endtikzpicture`.

Thus, she would use:

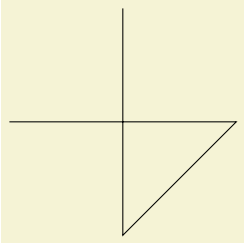
```
%% Plain TeX file
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda can typeset this file using either `pdftex` or `tex` together with `dvips`. Ti*k*Z will automatically discern which driver she is using. If she wishes to use `dvipdfm` together with `tex`, she either needs to modify the file `pgf.cfg` or can write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` somewhere *before* she inputs `tikz.tex` or `pgf.tex`.

3.3 Straight Path Construction

The basic building block of all pictures in Ti*k*Z is the path. A *path* is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in `(0,0)`.

This is followed by a series of "path extension commands." The simplest is `--`, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



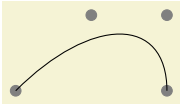
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl is a bit confused by the fact that there is no `{tikzpicture}` environment, here. Instead, the little command `\pgf` is used. This command either takes one argument (starting with an opening brace as in `\tikz{\draw (0,0) -- (1.5,0)}`, which yields _____) or collects everything up to the next semicolon and puts it inside a `{tikzpicture}` environment. As a rule of thumb, all Ti*k*Z graphic drawing commands must occur as an argument of `\tikz` or inside a `{tikzpicture}` environment. Fortunately, the command `\draw` will only be defined inside this environment, so there is little chance that you will accidentially do something wrong here.

3.4 Curved Path Construction

The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, Ti*k*Z provides a special syntax. One or two "control points" are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y. Then the curve will start "going in the direction y at x," that is, the tangent of the curve at x will point towards y. Next, suppose the curve should end at z and the second support point is w. Then the curve will, indeed, end in z and the tangent of the curve at point z will go through w.

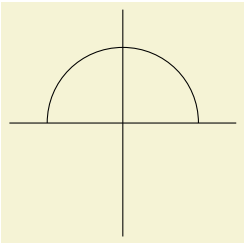
Here is an example (the control points have been added for clarity):



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

The general syntax for extending a path in a "curved" way is `.. controls` ⟨*first control point*⟩ `and` ⟨*second control point*⟩ `..` ⟨*end point*⟩. You can leave out the `and` and the ⟨*second control point*⟩, which causes the first one to be used twice.

So, Karl can now add the first half circle to the picture:



```
\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 is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

3.5 Circle Path Contruction

In order to draw a circle, the path construction command `circle` can be used. This command is followed by a radius in round brackets as in the following example: (Note that the previous position is used as the *center* of the circle.)

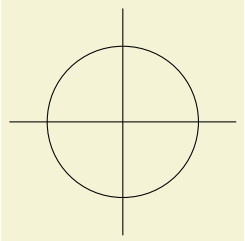
```
\tikz \draw (0,0) circle (10pt);
```

You can also append an ellipse to the path using the `ellipse` command. Instead of a single radius, you can specify two of them, one for the x-direction and one for the y-direction, seperated by a slash:

```
\tikz \draw (0,0) ellipse (20pt/10pt);
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbirtrary direction (a "turned ellipse" like ⬭) you can use transformations, which are explained later. The code for the little ellipse is `\tikz \draw[rotate=30] (0,0) ellipse (6pt/3pt);`, by the way.

So, returning to Karl's problem, he can write `\draw (0,0) circle (1cm);` to draw the circle:

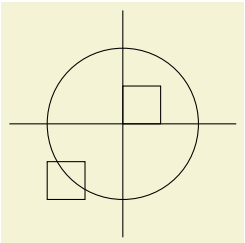


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that Ti*k*Z has powerful transformation commands and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

3.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to get it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the `rectangle` path construction command. This command should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:



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

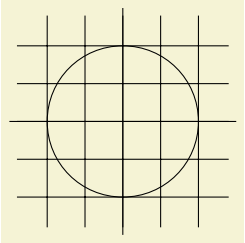
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed."

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `\draw` command, when he learns that there is a `grid` path construction command.

3.7 Grid Path Construction

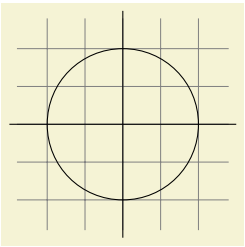
The `grid` path command adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` command. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces ▦. Note how the optional argument for `\draw` can be used to specify a grid width (there are also `xstep` and `ystep` to define the steppings independently). As Karl will learn soon, there are *lots* of things that can be influenced using such options.

For Karl, the following code could be used:



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

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the `\draw` command that draws the grid. First, he uses the color `gray` for the grid lines. Second, he reduced the line width to `very thin`. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\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 (1cm);
\end{tikzpicture}
```

3.8 Adding a Touch of Style

Instead of the options `gray,very thin` Karl could also have said `style=help lines`. *Styles* are predefined sets of options that can be used to organize how a graphic is drawn. By saying `style=help lines` you say "use the style that I (or someone else) has set for drawing help lines." If Karls decides, at some later point, that grids should be drawn, say, using the color `blue!50` instead of `gray`, he could say

```
\tikzstyle help lines=[color=blue!50,very thin]
```

Alternatively, he could have said

```
\tikzstyle help lines+=[color=blue!50]
```

which would have added the `color=blue!50` option. This would mean that the `help lines` style now contains *two* color options, but then the second overrides the first.

Using styles makes your graphics much more flexible since you can change the way things look easily in a consistent manner.

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

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

You can also leave out the `style=`. Thus, whenever Ti*k*Z encounters an options that it does not know about, it will check whether this option happens to be the name of a style. If so, the style is used. Thus, Karl could also have written:

```
\tikzstyle Karl's grid=[help lines,color=blue!50]
...
\draw[Karl's grid] (0,0) grid (5,5);
```

For some styles, like the `very thin` style, it is pretty clear what the style does and there is no need to say `style=very thin`. For other sytles, like `help lines`, it seems more natural to me to say `style=help lines`. But, mainly, this is a matter of taste.

3.9 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).

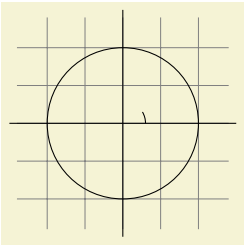
He saw that the option `very thin` yields very thin lines. Karl is not really suprised by this and neither is he suprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printes and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that `thin` is the correct choice. This seems strange to Karl, but his son explains him that LaTeX has two commands called `\thinlines` and `\thicklines` and that `\thinlines` gives the line width of "normal" lines, more precisely, of the thickness that, say, the stem of a letter like "T" or "i" has. 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 options `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 `closely 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.

3.10 Arc Path Construction

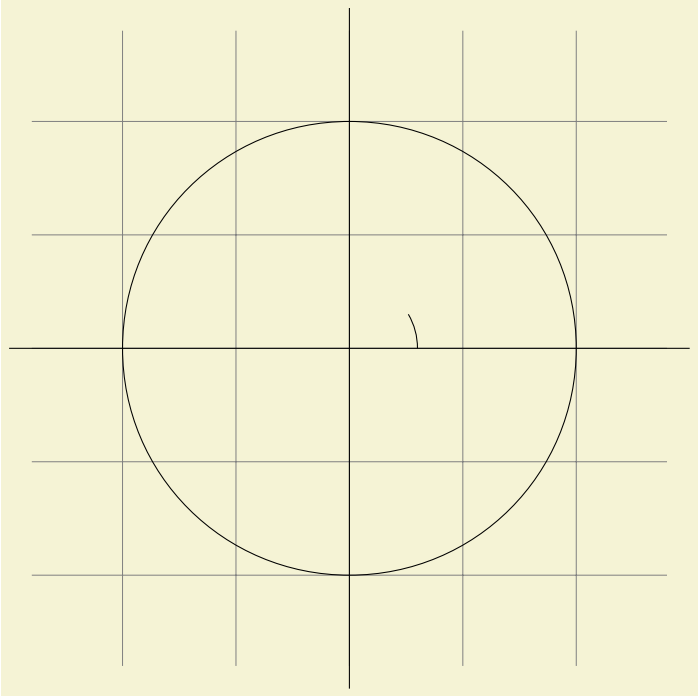
Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction command is useful, which draws part of a circle or ellipse. This `arc` command must be followed by a triple in rounded brackets, where the components of the triple are separated by colons. The first two components are angles, the last one is a radius. An example would be `(10:80:10pt)`, which means "an arc from 10 degrees to 80 degrees on a circle of radius 10pt." 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. This gives: `(0:30:3mm)`.

When one uses the arc path construction command, the specified arc will be added with its starting point at the current position. So, we first have to "get there."



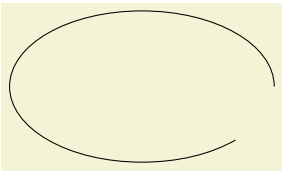
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

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 (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

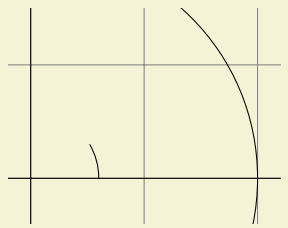
As for circles, you can specify "two" radii in order to get an elliptical arc.



```
  \tikz \draw (0,0) arc (0:315:1.75cm/1cm);
```

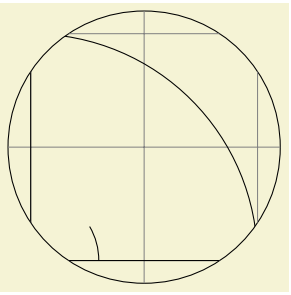
3.11 Clipping a Path

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 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 (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\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]` where "plain" stand for a "a plain path with which we would like to do something," and you could also say `\path[draw,clip]`.) Here is a nice 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 (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

3.12 Parabola and Sine Path Contruction

Although Karl does not need them for his picture, he is pleased to learn that there are `parabola` and `sin` and `cos` path commands for adding parabolas and sine and cosine curves to the current path. For the `parabola` command, the current point will lie on the parabola and the parabola will "fill" the rectangle whose corners are given by the current point and the point following the `parabola` command. Consider the following example:

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

There also exist `parabola left` and `parabola right` commands that yield only a left or a right side of a parabola. This can be used to draw only certain parts of a parabola:

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

The commads `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 exmples:

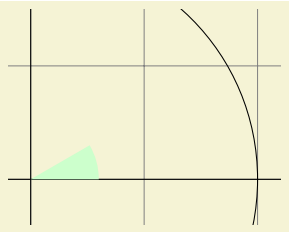
A sine ⌒ curve.

```
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);
```

3.13 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 (1cm);
  \fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- (0,0);
\end{tikzpicture}
```

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

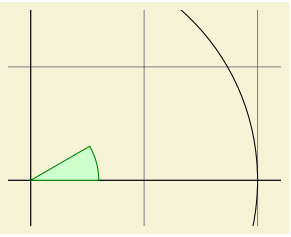
What would have happened, if Karl had not "closed" the path using `--(0,0)` at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

```
\fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
```

The `--cycle` causes the current path to be closed (actually the current segment 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;
\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 (1cm);
  \filldraw[fill=green!20!white, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

3.14 Shading

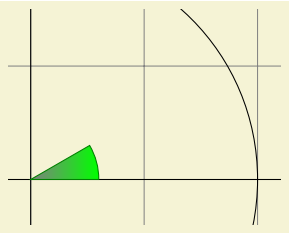
Karl briefly considers the possiblity of making the angle "more fancy" by *shading* it. Instead of filling the with a uniform color, a smooth transition between different colors is used. For this, the `\shade` and, for shading and drawing at the same time, `\shadedraw` can be used:

```
  \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=blue,bottom color=black] (0,0) rectangle +(2,1);
  \shade[left color=blue,right color=black] (3,0) rectangle +(2,1);
  \shadedraw[inner color=blue,outer color=black,draw=blue] (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 (1cm);
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

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

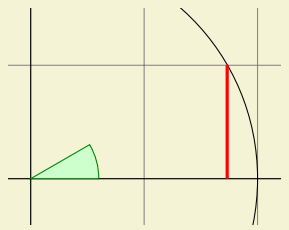
3.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `color=` option to set the lines's 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 x-direction and 1cm in 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."

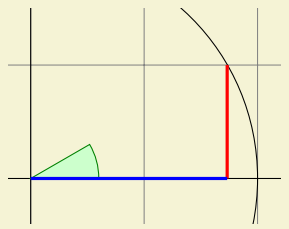
You can add a single `+` sign in front of a coordinate or two of them as in `+(1cm,0cm)` or `++(0cm,2cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making 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 (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- 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 `(`⟨p⟩ `|-` ⟨q⟩`)` is "the intersection of a vertical line through p and a horizontal line through q."

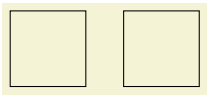
Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we "continue" from where the sine ends:



```
\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 (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]  (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

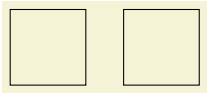
Note the there is no `--` between `(30:1cm)` and `+(0,-0.5)`. In detail, this path is interpreted as follows: "First, the `(30:1cm)` tells me to move by pen to $(\cos 30°, 1/2)$. Next, there comes another point specification so I move my pen there without drawing anything. This new point is half a unit 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:



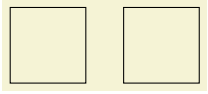
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

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 `+`):



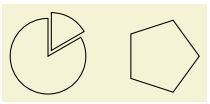
```
\tikz \draw (0,0) rectangle +(1,1)  (1.5,0) rectangle +(1,1);
```

Karl is left with the line for $\tan\alpha$, which seems difficult to specify using transformations and polar coordinates. So, he simply cheats: Using his computer's calculator he quickly finds out that $\tan 30° = 1/\sqrt{2} \approx 0.57735026919$. Since the accuracy of PGF and both PostScript and PDF is about 5 decimal digits, he will be just fine by saying

```
\draw[very thick,orange] (1,0) -- +(0,0.57735);
```

in order to draw the orange line.

In the following, two final examples of how to use relative positioning are presented. Note that the transformation commands, which are explained later, are often more useful for shifting than relative positioning.

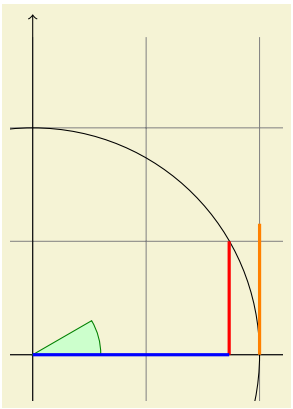


```
\begin{tikzpicture}[scale=0.5]
  \draw (0,0) -- (90:1cm) arc (90:360:1cm) arc (0:30:1cm) -- cycle;
  \draw (60:5pt) -- +(30:1cm) arc (30:90:1cm) -- cycle;

  \draw (3,0)  +(0:1cm) -- +(72:1cm) -- +(144:1cm) -- +(216:1cm) --
               +(288:1cm) -- cycle;
\end{tikzpicture}
```

3.16 Adding Arrows

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

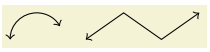
It turns out that adding arrows is pretty easy: Karl adds the option `->` to the drawing commands for the axes:



```
\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 (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]    (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick]   (30:1cm) ++(0,-0.5) -- (0,0);
  \draw[orange,very thick] (1,0) -- +(0,0.57735);
\end{tikzpicture}
```

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

There are certain restrictions to the kind of paths to which arrows can be added. As a rule of thumb, you can add arrows only to a single open "line." For example, you should not try to add arrows to, say, a rectangle or a circle. (You can try, but no guarantees as to what will happen now or in future versions.) However, you can add arrows to curved paths and to paths that have several segements as in the following examples:



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

Karl has a more detailed look at the arrow that Ti*k*Z 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 wish to use some other kinds of arrows.

To do so, Karl can say `>=`⟨*right arrow kind*⟩, where ⟨*right*⟩ is a special arrow specifications. For example, if Karl says `>=stealth-stealth`, then he tells Ti*k*Z that he would like "stealth-fighter-like" arrows:

```
\begin{tikzpicture}[>=stealth]
  \draw [->>] (0,0) arc (180:30:10pt);
  \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.

Instead of `stealth`, there are several other predefined arrow types Karl can choose from, see Section 11.1. Furthermore, he can define arrows types himself, if he needs new ones.

3.17 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`). 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`). 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. In 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 draw. 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.

3.18 Transformations

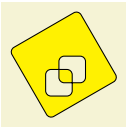
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, Ti*k*Z, TEX, and PDF all apply certain transformations to the given coordinate in order to determine the finaly position on the page.

Ti*k*Z provides numerous options that allow you to transform coordinates in PGF'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 normally not supported by PDF or PostScript. The reason is that PGF 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]    (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.

3.19 Repeating Things: For-Loops

Karl's next aim is to add little ticks on the axes at positions -1, -0.5, 0.5, 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 `\mulitdo` command. All of these can be used together with PGF and Ti*k*Z, so if you are familiar with them, feel free to use them. PGF 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 of PGF, but Ti*k*Z includes it automatically.

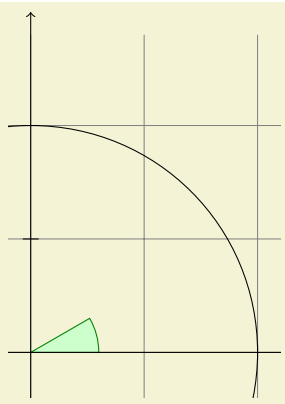
In its basic form, the `\foreach` command is easy to use:

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

```
\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
  (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (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 to create 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:

1,5	2,5	3,5	4,5	5,5		7,5	8,5	9,5	10,5	11,5	12,5
1,4	2,4	3,4	4,4	5,4		7,4	8,4	9,4	10,4	11,4	12,4
1,3	2,3	3,3	4,3	5,3		7,3	8,3	9,3	10,3	11,3	12,3
1,2	2,2	3,2	4,2	5,2		7,2	8,2	9,2	10,2	11,2	12,2
1,1	2,1	3,1	4,1	5,1		7,1	8,1	9,1	10,1	11,1	12,1

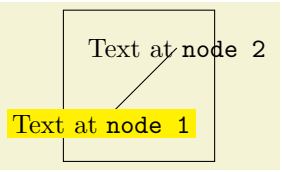
```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

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

3.20 Adding Text

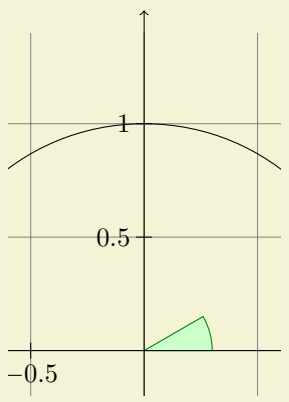
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

Ti*k*Z offers an easy-to-use and powerful system for adding text and, more generally, complex shapes at certain positions to a picture. 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 keywork `node` is typically followed by some option 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 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] {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 not necessarily *on* the last specified position, but rather 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 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 given 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:

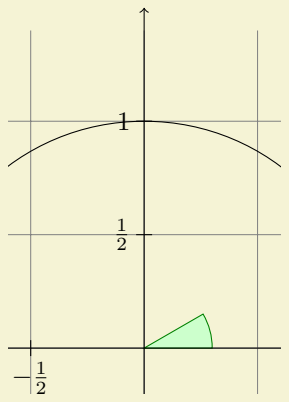



```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (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 east`. 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, very much prefer 0.5 over 1/2 since they are not too fond of fractions in general.

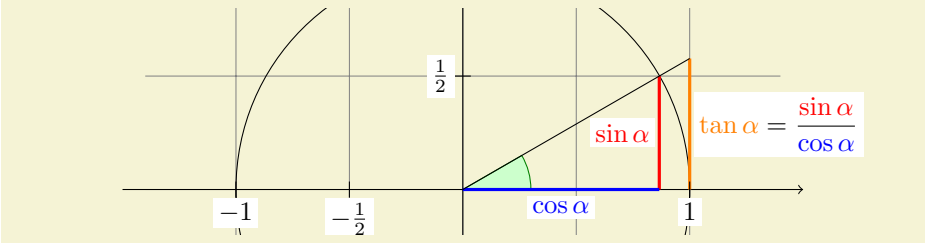
Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead 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,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (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}
```

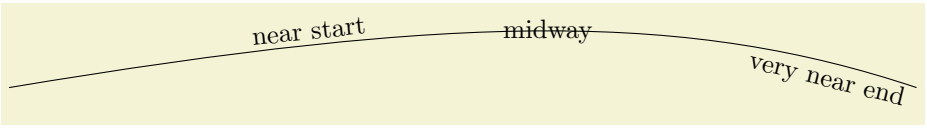
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy 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 line." To do so, instead of specifying the label `{$\sin\alpha$}` 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
  (0:30:3mm) -- 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 (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);
  \draw[very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
    \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} +(0,.577350);
  \draw (0,0) -- (1,0.57735);

  \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:



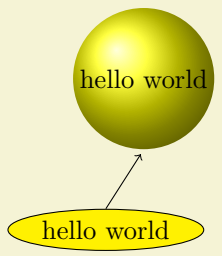

```
\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) ..
    node[near start,sloped,above] {near start}
    node {midway}
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}
```

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,cap=round]
  % Local definitions
  \def\costhirty{0.8660256}

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

  % Styles
  \tikzstyle{axes}=[]
  \tikzstyle{important line}=[very thick]
  \tikzstyle{information text}=[rounded corners,fill=red!10,inner sep=1ex]

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

  \draw (0,0) circle (1cm);

  \begin{scope}[style=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(0:30:3mm);
  \draw (15:2mm) node[anglecolor] {$\alpha$};

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

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

  \draw[style=important line,tancolor] (1,0) -- node[right=1pt,fill=white] {
    $\displaystyle \tan \alpha \color{black}=
    \frac{{\color{sincolor}\sin \alpha}}{\color{coscolor}\cos \alpha}$} +(0,.577350);

  \draw (0,0) -- (1,0.57735);

  \draw[xshift=1.85cm]
    node[right,text width=6cm,style=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 Phythagoras ...
    };
\end{tikzpicture}
```

3.21 Nodes

Placing text at a given position is just a special case of a more general underlying mechanism. When you say `\draw (0,0) node{text};`, what actually happens is that a rectangular node, anchored at its center, is put at position $(0, 0)$. On top of the rectangular node the text `text` is drawn. Since no action is specified for the rectangle (like `draw` or `fill`), the rectangle is actually discarded and only the text is shown. However, by adding `fill` or `draw`, we can make the underlying shape visible. Furthermore, we can *change* the shape using for example `shape=circle` or just `circle`. If we include the package `pgflibraryshapes` we also get `ellipse`:

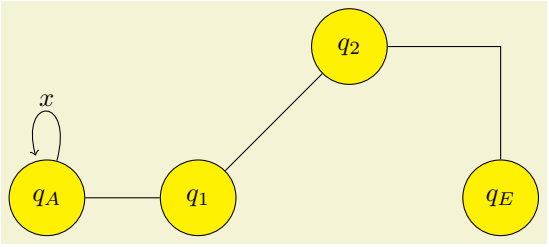



```
\begin{tikzpicture}
  \path (0,0)   node[ellipse,fill=yellow,draw]      (h1) {hello world}
        (0.5,2) node[circle,shade,ball color=yellow](h2) {hello world};
  \draw [->,shorten >=2pt] (h1.north) -- (h2.south);
\end{tikzpicture}
```

As the above example shows, we can add the a name to a node by putting it in parantheses at the very beginning (you can also use the `name=` option). This will make Ti*k*Z remember your node and all its anchors. You can then refer to these anchors when specifying coordinates. The syntax is `(`⟨*node name*⟩`.`⟨*anchor*⟩`)`. Currently, and also in the near future, *this will not work across pictures since Ti*k*Z looses track of the positions when it returns control to TEX.* Magic hackery is possible for certain drivers, but a portable implementation seems impossible (just think of a possible SVG driver).

The option `shorten >` causes lines to be shortened by 2pt at the end. Similarly, `shorten <` can be used to shorten (or even lengthen) lines at the beginning. This is possible even if no arrow is drawn.

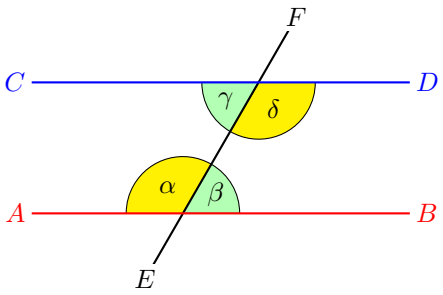
It is not always necessary to specify the anchor. If you do not give an anchor, Ti*k*Z will try to determine a reasonable border anchor by itself (if Ti*k*Z fails to find anything useful, it will use the center instead). Here is a typical example:




```
\begin{tikzpicture}
  \begin{scope}[shape=circle,minimum size=1cm,shape action={draw,fill},fill=yellow]
    \node (q_A) at (0,0) {$q_A$};
    \node (q_E) at (6,0) {$q_E$};
    \node (q_1) at (2,0) {$q_1$};
    \node (q_2) at (4,2) {$q_2$};
  \end{scope}
  \draw (q_A) -- (q_1) -- (q_2) -| (q_E);
  \draw[->,shorten >=2pt] (q_A) .. controls +(75:1.4cm) and +(105:1.4cm) .. node[above] {$x$} (q_A);
\end{tikzpicture}
```

In the example, we used the `\node` command, which is an abbreviation for `\path node`.

Part II
TikZ ist *kein* Zeichenprogramm



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

```
\begin{tikzpicture}
  \draw[fill=yellow] (0,0) -- (60:.75cm) arc (60:180:.75cm);
  \draw(120:0.4cm) node {$\alpha$};

  \draw[fill=green!30] (0,0) -- (right:.75cm) arc (0:60:.75cm);
  \draw(30:0.5cm) node {$\beta$};

  \begin{scope}[shift={(60:2cm)}]
    \draw[fill=green!30] (0,0) -- (180:.75cm) arc (180:240:.75cm);
    \draw (30:-0.5cm) node {$\gamma$};

    \draw[fill=yellow] (0,0) -- (240:.75cm) arc (240:360:.75cm);
    \draw (-60:0.4cm) node {$\delta$};
  \end{scope}

  \begin{scope}[thick]
    \draw  (60:-1cm) node[fill=white] {$E$} -- (60:3cm) node[fill=white] {$F$};
    \draw[red]                    (-2,0) node[left] {$A$} -- (3,0) node[right]{$B$};
    \draw[blue,shift={(60:2cm)}] (-3,0) node[left] {$C$} -- (2,0) node[right]{$D$};

    \draw[shift={(60:1cm)},xshift=4cm]
    node [right,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{scope}
\end{tikzpicture}
```

4 Design Principles

This section describes the design principles behind the Ti*k*Z frontend, where Ti*k*Z means "Ti*k*Z ist *kein* Zeichenprogramm." To use Ti*k*Z, as a LaTeX user say `\usepackage{tikz}` somewhere in the preamble, as a plain TeX user say `\input tikz.tex`. Ti*k*Z's job is to make your life easier by providing a easy-to-learn and easy-to-use syntax for describing graphics.

Users familiar with METAFONT will find that the commands and the syntax of Ti*k*Z often resembles METAFONT. The commands and syntax of PSTRICKS has also influenced Ti*k*Z's syntax. However, in some cases I decided to use a new syntax that, I hope, is more appropriate than the syntax of either METAFONT or PSTRICKS.

The following basic design principles underlie Ti*k*Z:

1. Special syntax for specifying points.
2. Special syntax for path specifications.
3. Actions on paths.
4. Key-value syntax for graphic parameters.
5. Special syntax for nodes.
6. Grouping of graphic parameters.
7. Coordinate transformation system.

4.1 Special Syntax For Specifying Points

Ti*k*Z provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TeX dimensions, separated by commas, in round brackets as in `(1cm,2pt)`.

You can also specify a point in polar coordinates by using a colon instead of a comma as in `(30:1cm)`, which means "1cm in a 30 degrees direction."

If you do not provide a unit, as in `(2,1)`, you specify a point in PGF's xy-coordiate system. By default, the unit x-vector goes 1cm to the right and the unit y-vector goes 1cm upward.

By specifying three numbers as in `(1,1,1)` you specify a point in PGF's xyz-coordinate system.

It is also possible to use an anchor of a previously defined shape as in `(first node.south)`.

You can add two plus signs before a coordinate as in `++(1cm,0pt)`. This means "1cm to the right of the last point used." This allows you to easily specify relative movements. For example, `(1,0) +(1,0) +(0,1)` specifies the three coodinates `(1,0)`, then `(2,0)`, and `(2,1)`.

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not "change" the current point used in subsequent relative commands. For example, `(1,0) +(1,0) +(0,1)` specifies the three coodinates `(1,0)`, then `(2,0)`, and `(1,1)`.

4.2 Special Syntax For Path Specifications

When creating a picture using Ti*k*Z, your main job is the creation of *paths*. A path is a series of straight or curved lines, which need not be connected. Ti*k*Z makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get ◺ when you draw this path.

4.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can *draw* a path, *fill* a path, *shade* it, *clip* it, or do any combination of these. Drawing (also known as *stroking*) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for *closed* paths and a path is automatically closed prior to filling, if necessary.

Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields ▭. The `\draw` command is just an abbreviation

for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation for `\path[fill]`. The `\filldraw` command is an abbreviation for `\path[fill,draw]`. Shading is caused by the `shade` option (there are `\shade` and `\shadedraw` abbreviations) and clipping by the `clip` option. There is is also a `\clip` command, which does the same as `\path[clip]`, but not commands like `\drawclip`. Use, say, `\draw[clip]` or `\path[draw,clip]` instead.

All of these commands can only be used inside `{tikzpicture}` environments.

Ti*k*Z allows you to use different colors for filling and stroking.

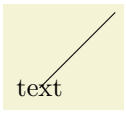
4.4 Key-Value Syntax for Graphic Parameters

Whenever Ti*k*Z draws or fills a path, a large number of graphic parameters influenced the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In Ti*k*Z, all these options are specified as lists of so called key-value pairs, as in `color=red`, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;


```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (1,0) -- cycle;
```

4.5 Special Syntax for Specifcying Nodes

Ti*k*Z introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, you can insert some normal TEX text surrounded by curly braces as in




```
\tikz \draw (1,1) node {text} -- (2,2);
```

The text in curly braces will be inserted at the current position of the path, but only *after* the path has been rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and this circle is "drawn."

You can add a name to a node for later reference either by using the option `name=`⟨*node name*⟩ or by stating the node name in parantheses outside the text as in `node[cricle](name){text}`.

Predefined shapes include `rectangle`, `circle`, and `ellipse`, but it is possible (though a bit challenging) to define new shapes.

4.6 Grouping of Graphic Parameters

Graphic parameters often need to apply to several path drawing or filling commands. For example, we may wish to numerous lines all with the same line width of 1pt. For this, we put these commands in a `{scope}` environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested `{scope}` environments or individual drawing commands can override the graphic parameters or outer `{scope}` environments. In the following example, three red lines, two green lines, and one blue line are drawn:

```
\begin{tikzpicture}
  \begin{scope}[color=red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm, 8mm) -- (10mm, 8mm);
    \draw (0mm, 6mm) -- (10mm, 6mm);
  \end{scope}
  \begin{scope}[color=green]
    \draw (0mm, 4mm) -- (10mm, 4mm);
    \draw (0mm, 2mm) -- (10mm, 2mm);
    \draw[color=blue]
          (0mm, 0mm) -- (10mm, 0mm);
  \end{scope}
\end{tikzpicture}
```

The `{tikzpicture}` environment itself also behaves like a `{scope}` environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

4.7 Coordinate Transformation System

Ti*k*Z relies entirely on PGF's *coordinate* transformation system to perform transformations. PGF also supports *canvas* transformations, a more low-level transformation system, but this system is not accessible from Ti*k*Z. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF looses track of where nodes and shapes are positioned when canvas transformations are used.

For more details on the difference between coordinate transformations and canvas transformations see Section 15.4.

5 Hierarchical Structures: Package, Environments, Scopes, and Styles

The present section explains how your files should be structured when you use Ti*k*Z. On the top level, you need to include the `tikz` package. In the main text, each graphi needs to be put in a `{tikzpicture}` environment. Inside these environments, you can use `{scope}` environments to create internal groups. Inside the scopes, you use `\path` commands to actually draw something. On all levels (except for the package level), graphic options can be given that apply to everything within the environment.

5.1 Loading the Package

```
\usepackage{tikz} % LaTeX
\input tikz.tex   % plain TeX
\input tikz.tex   % ConTeX
```

This package does not have any options.

This will automatically load the PGF package and several other stuff that Ti*k*Z needs (like the `xkeyval` package).

PGF needs to know what TEX driver you are intending to use. In most cases, PGF is clever enough to find out the correct driver for you; this is true in particular if you LATEX. Currently, the only situation where PGF cannot know the driver "by itself" is when you use plain TEX or ConTEXt together with `dvipdfm`. In this case, you have to write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` *before* you input `tikz.tex`.

5.2 The Main Picture Environment

The "outermost" scope of PGF and Ti*k*Z is the `{tikzpicture}` environment. You may give drawing commands only inside this environment, giving them outside (as is possible in many other packages) will result in chaos.

In Ti*k*Z, most of how a graphic "looks like" is governed by graphic options. For example, there is an option for setting the color used for drawing, another for setting the color used for filling, and some more obscure one like the option for setting the prefix used in the filenames of temporary files written while plotting functions using an external program. The graphic options are nearly always specified in a so-called key-value style. (The "nearly always" refers to the name of nodes, which can also be specified differently.) All graphic options are local to the `{tikzpicture}` to which they apply.

`\begin{tikzpicture}`[⟨*options*⟩]
⟨*environment contents*⟩
`\end{tikzpicture}`

All Ti*k*Z and nearly all PGF commands should be given inside this environment. Unlike other packages, it is not possible to use, say, `\pgfpathmoveto` outside this environment and will result in chaos. For Ti*k*Z, commands like `\path` are only defined inside this environment, so there is little chance that you will do something wrong here.

When this environment is encountered, the ⟨*options*⟩ are parsed. All options given here will apply to the whole picture. Giving options makes sense only when Ti*k*Z is loaded, the PGF core does not define any options.

Next, the contents of the environment is processed and the graphic commands therein are put into a box. Non-graphic text is suppressed as well as possible, but non-PGF commands inside a `{tikzpicture}` environment should not produce any "output" since this may totally scramble the positioning system of the backend drivers. The suppressing of normal text, by the way, is done by temporarily switching the font to `\nullfont`. You can, however, "escape back" to normal TEX typesetting. This happens, for example, when you specify a node.

At the end of the environment, PGF tries to make a good guess at a good guess at the size of the bounding box of the graphic and then resizes the box such that the box has this size. To "make its guess," everytime PGF encounters a coordinate, it updates the bound box's size such that it encompasses all these coordinates. This will usually give a good approximation at the bounding box, but will not always be accurate. First, the line thickness is not taken into account. Second, controls points of a

curve often lie far "outside" the curve and make the bounding box too large. In this case, you should use the `[use as bounding box]` option.

The following option influences the baseline of the resulting picture:

- `baseline=`⟨*dimension*⟩ Normally, the lower end of the picture is put on the baseline of the surrounding text. For example, when you give the code `\tikz\draw(0,0)circle(.5ex);`, PGF will find out that the lower end of the picture is at −.5ex and that the upper end is at .5ex. Then, the lower end will be put on the baseline, resulting in the following: ○.

  Using this option, you can specify that the picture should be raised or lowered such that the height ⟨*dimension*⟩ is on the baseline. For example, `tikz[baseline=0pt]\draw(0,0)circle(.5ex);` yields ○ since, now, the baseline is on the height of the x-axis. If you omit the ⟨*dimensions*⟩, `0pt` is assumed as default.

  This options is often useful for "inlined" graphics as in

  $A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$ `$A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$`

All options "end" at the end of the picture. In particular, there is no official way of globally setting options as with the `\psset` command of PSTRICKS. There are three reasons for this:

1. Global settings make images "less portable." If every picture "carries around" all its options, it will be much easier to reuse a graphic in another document or even just in another section.
2. You can use styles to set options consistently in a clear, "portable" way.
3. You can cheat by saying `\setkeys{tikz}{`metaglobal option`}`.

In plain TEX, you should use instead the following commands:

`\tikzpicture[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\endtikzpicture`

The following two commands are used for "small" graphics.

`\tikz[`⟨*options*⟩`]{`⟨*commands*⟩`}`

This little command places the ⟨*commands*⟩ inside a `{tikzpicture}` environment and adds a semicolon at the end. This is just a convenience.

The ⟨*commands*⟩ may not contain a paragraph (an empty line). This is a precaution to ensure that users really use this command only for small graphics.

Example: `\tikz{\draw (0,0) rectangle (2ex,1ex)}` yields ▭

`\tikz[`⟨*options*⟩`]`⟨*text*⟩`;`

If the ⟨*text*⟩ does not start with an opening brace, the end of the ⟨*text*⟩ is the next semicolon that is encountered.

Example: `\tikz \draw (0,0) rectangle (2ex,1ex);` yields ▭

5.3 Scopes

Inside a `{tikzpicture}` environment, you can create "subscopes" using the `{scope}` environment. This environment is available only inside the `{tikzpicture}` environment, so once more, there is little chance of doing anything wrong.

`\begin{scope}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{scope}`

All ⟨*options*⟩ are local to the ⟨*environment contents*⟩. Furthermore, the clipping path is also local to the environment, that is, any clipping done inside the environment "ends" at its end.

Example: 

```
\begin{tikzpicture}
  \begin{scope}[red]
    \draw (0mm,5mm) -- (10mm,5mm);
    \draw (0mm,4mm) -- (10mm,4mm);
  \end{scope}
  \draw (0mm,3mm) -- (10mm,3mm);
  \begin{scope}[green]
    \draw (0mm,2mm) -- (10mm,2mm);
    \draw (0mm,1mm) -- (10mm,1mm);
    \draw[blue] (0mm,0mm) -- (10mm,0mm);
  \end{scope}
\end{tikzpicture}
```

In plain TEX, you use the following commands instead:

```
\scope[⟨options⟩]
⟨environment contents⟩
\endscope
```

5.4 Path Scopes

The `\path` command, which is described in much more detail in later sections, also takes graphic options. These options are local to the path. Furthermore, it is possible to create local scopes withing a path simply by using curly braces as in

```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

Note that many options apply only to the path as a whole and cannot be scoped in this way. For example, it is not possible to scope the `color` of the path. See the explanations in the section on paths for more details.

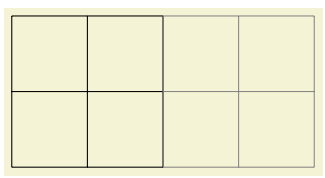
Finally, certain elements that you specify in the argument to the `\path` command also take local options. For example, a node specification takes options. In this case, the options apply only to the node, not to the surrounding path.

5.5 Styles

There is a way of organizing sets of graphic options "orthogonally" to the normal scoping mechanism. For example, you might wish all your "help lines" to be drawn in a certain way like, say, gray and thin (do *not* dash them, that distracts). For this, you can use *styles*.

A style is simply a set of graphic options that is predefined at some point. Once a style has been defined, it can be used anywhere using the `style` option:

- `style=`⟨*style name*⟩ invokes all options that are currently set in the ⟨*style name*⟩. An example of a style is the predefined `help lines` style, which you should use for lines in the background like grid lines or construction lines. You can easily define new styles and modify existing ones.

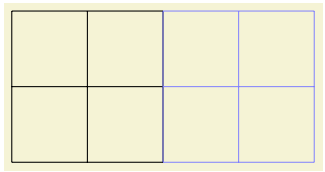


```
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

`\tikzstyle`⟨*style name*⟩`+=[`⟨*options*⟩`]`

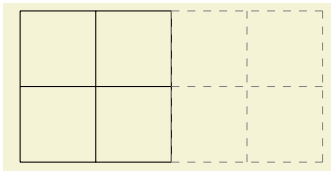
This command defines the style ⟨*style name*⟩. Whenever it is used using the `style=`⟨*style name*⟩ command, the ⟨*options*⟩ will be invoked. It is permissible that a style invokes another style using the `style=` command inside the ⟨*options*⟩, which allows you to build hierarchies of styles. Naturally, you should *not* create cyclic dependencies.

If the style already has a predefined meaning, it will uncermimoniously be redefined without a warning.



```
\tikzstyle help lines=[blue!50,very thin]
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

If the optional `+` is given, the options are *added* to the existing defintion:



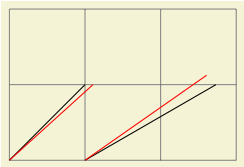
```
\tikzstyle help lines+=[dashed] % aaarghhh!!!
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

6 Specifying Coordinates

6.1 Coordinates and Coordinate Options

A *coordinate* is a position in a picture. Ti*k*Z uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is `(`[⟨*options*⟩]⟨*coordinate specification*⟩`)`.

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only. To give transformation options for a single coordinate, give these options at the beginning in brackets:



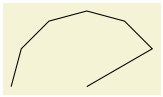
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=3pt] 1,1);
  \draw      (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```

6.2 Simple Coordinates

The simplest way is to specify as a comma-separated pair of TEX dimensions as in `(1cm,2pt)`. As can be seen, different units can be mixed. The coordinate specified in this way means "1cm to the right and 2pt up from the origin of the picture." You can also write things like `(1cm+2pt,2pt)` since the `calc` package is used, internally.

6.3 Polar Coordinates

You can also specify coordinates in polar coordinates. In this case, you specify an angle and a distance, separated by a colon as in `(30:1cm)`. The angle must always be given in degrees and should be between −360 and 720.



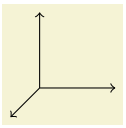
```
\tikz \draw    (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (90:1cm)
            -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Instead of an angle given as a number you can also use certain words. For example, `up` is the same as `90`, so that you can write `\tikz \draw (0,0) -- (2ex,0pt) -- +(up:1ex);` and get ⌊. Apart from `up` you can use `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east`, `north west`, `south east`, `south west`, all of which have their natural meaning.

6.4 xy-, and xyz-Coordinates

Next, you can specify coordinates in PGF's xy-coordinate system. In this case, you provide two unit-free numbers, separated by a comman as in `(2,-3)`. This means "add twice the current PGF x-vector and subtract three times the y-vector." By default, the x-vector points 1cm to the right, the y-vector points 1cm upwards, but this can be changed arbitrarily using the `x` and `y` graphic options.

Finally, you can specify coordinate in the xyz-coordinate system. The only difference to the xy-coordinates is that you specify three numbers separated by commas as in `(1,2,3)`. This is interpreted as "once the x-vector plus twice the y-vector plus three times the z-vector." The default z-vector points to $\left(-\frac{1}{\sqrt{2}}\text{cm}, -\frac{1}{\sqrt{2}}\text{cm}\right)$. Consider the following example:



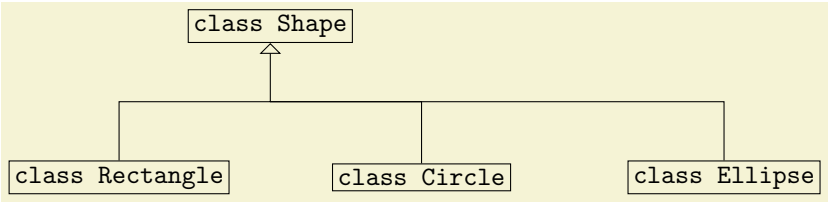
```
\begin{tikzpicture}[->]
  \draw (0,0,0) -- (1,0,0);
  \draw (0,0,0) -- (0,1,0);
  \draw (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

6.5 Node Coordinates

In PGF and in Ti*k*Z it is quite easy to define a node that you wish to reference at a later point. Once you have defined a node, there are different ways of referencing points of the node.

6.5.1 Named Anchor Coordinates

An *anchor coordiante* is a point in a node that you have previously defined using the curly braces syntax. The syntax is `(`⟨*node name*⟩`.`⟨*anchor*⟩`)`, where ⟨*node name*⟩ is the name that was previously used to name the node using the `name=`metanode name option or the special node name syntax. Here is an example:

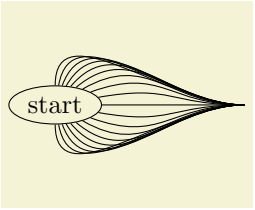



```
\begin{tikzpicture}
  \node (shape)   at (0,2)  [draw] {|class Shape|};
  \node (rect)    at (-2,0) [draw] {|class Rectangle|};
  \node (circle)  at (2,0)  [draw] {|class Circle|};
  \node (ellipse) at (6,0)  [draw] {|class Ellipse|};

  \draw (circle.north) |- (0,1);
  \draw (ellipse.north) |- (0,1);
  \draw[-open triangle 90] (rect.north) |- (0,1) -| (shape.south);
\end{tikzpicture}
```

Section 9.7 explain which anchors are available for the basic shapes.

6.5.2 Angle Anchor Coordinates

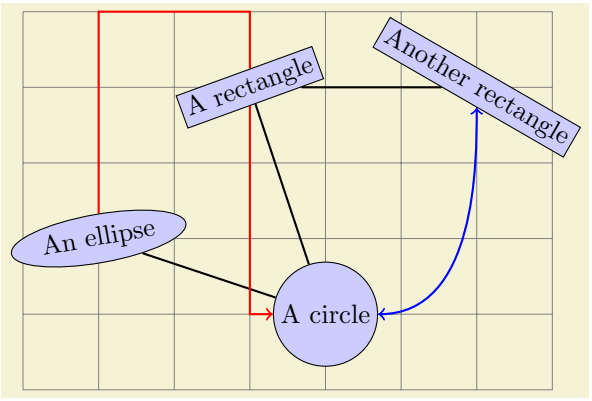
In addition to the named anchors, it is possible to use the syntax ⟨*node name*⟩`.`⟨*angle*⟩ to name a point of the node's border. This point is the coordinate where a ray shot from the center in the given angle hits the border. Here is an example:




```
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
    \draw (start.\angle) .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
  \end{tikzpicture}
```

6.5.3 Anchor-Free Node Coordinates

It is also possible to just "leave out" the anchor and have Ti*k*Z calculate an appropriate border position for you. Here is an example:



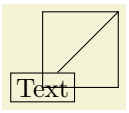
```
\begin{tikzpicture}[fill=blue!20]
  \draw[style=help lines] (-1,-2) grid (6,3);
  \path (0,0)  node(a) [ellipse,rotate=10,draw,fill]    {An ellipse}
        (3,-1) node(b) [circle,draw,fill]               {A circle}
        (2,2)  node(c) [rectangle,rotate=20,draw,fill]  {A rectangle}
        (5,2)  node(d) [rectangle,rotate=-30,draw,fill] {Another rectangle};
  \draw[thick] (a) -- (b) -- (c) -- (d);
  \draw[thick,red,->] (a) |- +(1,3) -| (c) |- (b);
  \draw[thick,blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

TikZ will be reasonably clever at determining the border points that you "mean," but, naturally, this may fail in some siutations. If TikZ fails to determine an appropriate border point, the center will be used, instead.

Automatic computation of anchors works only with the lineto command `--`, the vertical/horizontal versions `|-` and `-|`, and with the curveto command `...` For other path commands such as `parabola` or `plot`, the center will be used. If this is not desired, you should give a named anchor or an angle anchor.

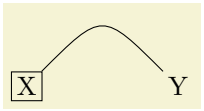
Note that if you use an automatic coordinate for both the start and the end of a lineto, as in `--(b)--`, then *two* border coordinates are computed with a moveto between them. This is usually exactly what you want.

If you use relative coordinates together with automatic anchor coordinates, the relative coordinates are always computed relative to the node's center, not relative to the border point. Here is an example:




```
\tikz \draw (0,0) node(x) [draw] {Text}
            rectangle (1,1)
            (x) -- +(1,1);
```

Similarly, the control points in the following examples both control points are $(1, 1)$:




```
\tikz \draw (0,0) node(x) [draw] {X}
            (2,0) node(y) {Y}
            (x) .. controls +(1,1) and +(-1,1) .. (y);
```

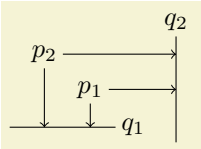
6.5.4 Intersection Coordinates

Often you wish to specify a point that is on the intersection of a vertical line going through a point p and a horizontal line going through some other point q.

There are two ways of specifying this intersection. The first is `(`⟨p⟩ `|-` ⟨q⟩`)`, the second is `(`⟨q⟩ `-|` ⟨p⟩`)`.

For example, `(2,1 |- 3,4)` and `(3,4 -| 2,1)` both yield the same as `(2,4)` (provided the xy-coordinate system has not been modified).

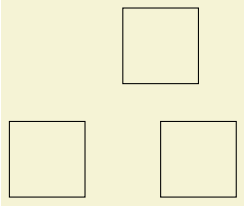
The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:




```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$}   (75:1cm) node(p2) {$p_2$};

  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};

  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

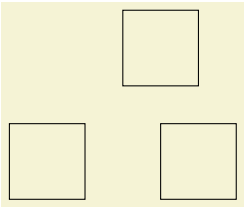
6.6 Relative and Incremental Coordinates

You can prefix coordinates by `++` to make them "relative." A coordinate such as `++(1cm,0pt)` means "1cm to the right of the previous position." Relative coordinates are often useful in "local" contexts:



```
\begin{tikzpicture}
  \draw (0,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (2,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (1.5,1.5) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\end{tikzpicture}
```

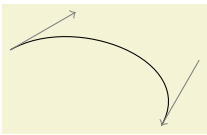
Intead of `++` you can also use a single `+`. This also specifies a relative coordinate, but it does not "update" the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same "initial" point:



```
\begin{tikzpicture}
  \draw (0,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (2,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

There is one special situation, where relative coordinates are interpreted differently. If you use a relative coordinate as a control point of a Bézier curve, the following rule applies: First, a relative first control point is taken relative to the beginning of the curve. Second, a relative second control point is taken relative to the end of the curve. Third, a relative end point of a curve is taken relative to the start of the curve.

This special behaviour makes it easy to specify that a curve should "leave or arrives from a certain direction" at the start or end. In the following example, the curve "leaves" at 30° and "arrives" at 60°:



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

7 Syntax for Path Specifications

A *path* is a series of straight and curved line segments. It is specified following a `\path` command and the specification must follow a special syntax, which is described in the subsections of the present section.

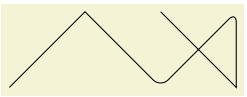
`\path`⟨*specification*⟩`;`

This command is available only inside a `{tikzpicture}` environment and only if the Ti*k*Z package is loaded.

The ⟨*specification*⟩ is a long stream of *path operations*. Most of these path operations tell Ti*k*Z how the path is build. For example, when you write `--(0,0)`, you use a *lineto operation* and it means "continue the path from wherever you are to the origin."

At any point where Ti*k*Z expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as `[rounded cornders]`. These options can have different effects:

1. Some options take "immediate" effect and apply to all subsequent path operations on the path. For example, the `rounded corners` option will round all following corners, but not the corners "before" and if the `sharp corners` is given later on the path (in a new set of brackets), the rounding effect will end.



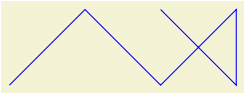
```
\tikz \draw (0,0) -- (1,1)
          [rounded corners] -- (2,0) -- (3,1)
          [sharp corners] -- (3,0) -- (2,1);
```

   Another example are the transformation options, which also apply only to subsequent coordinates.

2. The options that have immediate effect can be "scoped" by putting part of a path in curly braces. For example, the above example could also be written as follows:

```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

3. Some options only apply to the path as a whole. For example, the `color=` option for determining the color used for, say, drawing the path always applies to all parts of the path. If several different colors are given for different parts of the path, only the last one (on the outermost scope) "wins":



```
\tikz \draw (0,0) -- (1,1)
          [color=red] -- (2,0) -- (3,1)
          [color=blue] -- (3,0) -- (2,1);
```

   Most options are of this type. In the above example, we would have had to "split up" the path into several `\path` commands:

```
\tikz{\draw (0,0) -- (1,1);
      \draw [color=red] (2,0) -- (3,1);
      \draw [color=blue] (3,0) -- (2,1);}
```

By default, the `\path` command does "nothing" with the path, it just "throws it away." Thus, if you write `\path(0,0)--(1,1);`, nothing is drawn in your picture. The only effect is that the area occupied by the picture is (possibly) enlarged so that the path fits inside the area. To actuall "do" something with the path, an option like `draw` or `fill` must be given somewhere on the path. Commands like `\draw` do this implicitly.

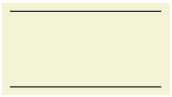
Finally, it is also possible to give *node specifications* on a path. Such specifications can come at different locations, but they are always allowed when a normal path operation could follow. A node specification starts with `node`. Basically, the effect is to typeset the node's text as normal TEX text and to place it at the "current location" on the path. The details are exlained in Section 9.

Note, however, that the nodes are *not* part of the path in any way. Rather, after everything has been done with the path what is specified by the path options (like filling and drawing the path due to a `fill` and a `draw` option somewhere in the ⟨*specification*⟩), the nodes are added in a post-processing step.

7.1 The Move-To Operation

A *move-to operation* normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a new part of the path that is not connected to any of the previous segments.

The syntax for specifying a move-to operation is easy: ⟨*coordinate*⟩. Simply provide a coordinate in round brackets. For example, the following command (when used inside a `{tikzpicture}` environment) draws two lines:



```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

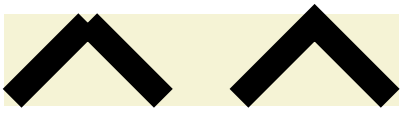
In the specification `(0,0) --(2,0) (0,1) --(2,1)` two move-to operations are specified: `(0,0)` and `(0,1)`. The other two operations, namely `--(2,0)` and `--(2,1)` are line-to operations, described next.

7.2 The Line-To Operation

A *line-to operation* extends the current path from the current point in a straight line to the given coordinate. The "current point" is the endpoint of the previous drawing command or the point specified by a prior moveto operation.

Syntax of the line-to operation: `--`⟨*coordinate*⟩. You use two minus signs followed by a coordinate in round brackets. You can add spaces before and after the `--`.

When a line-to operation is used and some path segment has just been constructed, for example by another line-to operation, the two line segments become joined. This means that if they are drawn, the point where they meet is "joined" smoothly. To appreciate the difference, consider the following two examples: In the left example, the path consists of two path segments that are not joined, but that happen to share a point, while in the right example a smooth join is shown.



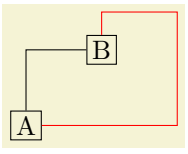
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1)  (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
\end{tikzpicture}
```

7.3 Horizontal/Vertical Line-To Operations

Sometimes you want to connect two points via straight lines that are only horizontal and vertical. For this, you can use the path construction commands `-|` and `|-`. The first means "first horizontal, then vertical," the second means "first vertical, then horizontal."

Syntax of these versions of the line-to operation: `-|`⟨*coordinate*⟩ and `|-`⟨*coordinate*⟩. You can add spaces for clarity.

Here is an example:




```
\begin{tikzpicture}
  \draw (0,0) node(a) [draw] {A}  (1,1) node(b) [draw] {B};
  \draw (a.north) |- (b.west);
  \draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

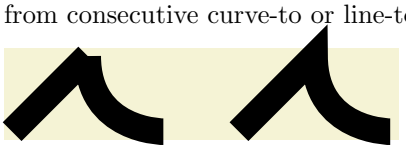
7.4 The Curveto Operation

A *curveto operation* extends the current path from the current point, let us call it x, via a curve to a new current point, let us call it y. The curve is a so-called Beziér curve. For such a curve, apart from y, you also specify two control points c and d. The idea is that the curve starts at x, "heading" in the direction of c. Mathematically spoken, the tangent of the curve at x goes through c. Similarly, the curve ends at y, "coming from" the other control point, d. The larger the distance between x and c and between d and y, the larger the curve will be.

Syntax of the line-to operation: `..controls`⟨*c*⟩`and`⟨*d*⟩`..`⟨*y*⟩. If ⟨*d*⟩ is not given, *d* is assumed to be equal to *c*.

```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) .. controls (1,1) .. (4,0)
                             .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=gray] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```

As with the line-to operation, it makes a difference whether two curves are joined because they resulted from consecutive curve-to or line-to operations, or whether they just happen to have the same ending:



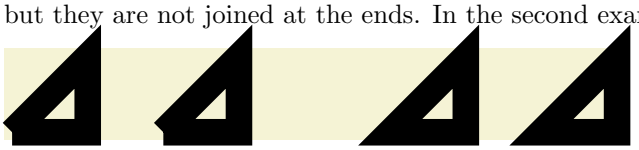
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw (3,0) -- (4,1) .. controls (4,0) and (5,0) .. (5,0);
\end{tikzpicture}
```

7.5 The Cycle Operation

A *cycle operation* adds a straight line from the current point to the last point specified by a move-to operation. Note that this need not be the beginning of the path. Furthermore, a smooth join is created between the first segment created after the last move-to operation and the straight line appended by the cycle operation.

Syntax of the line-to operation: `--cycle`. You can add a space between `--` and `cycle` for clarity.

Consider the following example. In the left example, two triangles are created using three straight lines, but they are not joined at the ends. In the second example cycle operations are used.

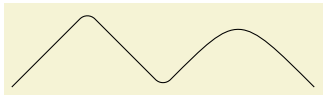


```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
  \draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
\end{tikzpicture}
```

7.6 Rounding Corners

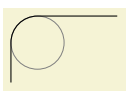
All of the path construction commands mentioned up to now are influenced by the following option:

- `rounded corners=`⟨*inset*⟩ When this option is in force, all corners (places where a line is continued either via line-to or a curve-to operation) are replaced by little arcs so that the corner becomes smooth.



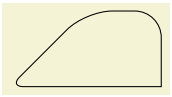
```
\tikz \draw [rounded corners] (0,0) -- (1,1)
          -- (2,0) .. controls (3,1) .. (4,0);
```

The ⟨*inset*⟩ describes how big the corner is. Note that the ⟨*inset*⟩ is *not* scaled along if you use a scaling option like `scale=`.



```
\begin{tikzpicture}
  \draw[color=gray,very thin] (10pt,15pt) circle (10pt);
  \draw[rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
\end{tikzpicture}
```

You can switch the rounded corners on and off "in the middle of path" and different corners in the same path can have different corner radii:



```
\begin{tikzpicture}
  \draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
                    [sharp corners] -- (2,0)
                [rounded corners=5pt] -- cycle;
\end{tikzpicture}
```

Example: Here is a rectangle with rounded corners:

```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

You should be aware, that there are several pitfalls when using this option. First, the rounded corner will only be an arc (part of a circle) if the angle is 90°. In other cases, the rounded corner will still be round, but "not as nice."

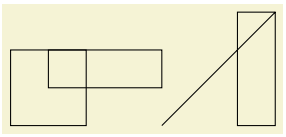
Second, if there are very short line segements in a path, the "rounding" may cause inadverted effects. In such case it may be necessary to temporarily switch off the rounding using `sharp corners`.

- `sharp corners` This options switches off any rounding on subsequent corners of the path.

7.7 The Rectangle Operation

A rectangle can obviously be created using four straight lines and a cycle operation. However, since rectangles are needed so often, a special syntax is available for them. When the rectangle operation is used, one corner will be the current point, another corner will be given by the specified point. The specified point becomes the new current point.

Syntax of the rectanlge operation: `rectangle`⟨*corner*⟩. You can add a space.



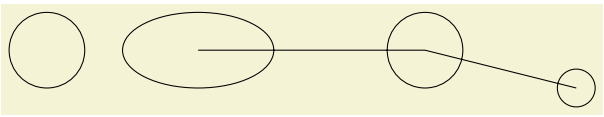
```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,1);
  \draw (.5,1) rectangle (2,0.5) (3,0) rectangle (3.5,1.5) -- (2,0);
\end{tikzpicture}
```

7.8 The Circle and Ellipse Operations

A circle can be approximated well using four Beziér curves. However, it is difficult to do so correctly. For this reason, a special syntax is available for adding such an approximation of a circle to the current path. The center of the circle is given by the current point. The new current point of the path will remain to be the center of the circle.

Syntax of the circle operation: `circle(`⟨*radius*⟩`)`. You can add spaces.

Syntax of the ellipse operation: `ellipse(`⟨*half width*⟩`/`⟨*half height*⟩`)`. You can add spaces.

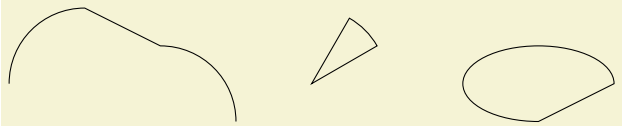


```
\begin{tikzpicture}
  \draw (1,0) circle (.5cm);
  \draw (3,0) ellipse (1cm/.5cm) -- ++(3,0) circle (.5cm) -- ++(2,-.5) circle (.25cm);
\end{tikzpicture}
```

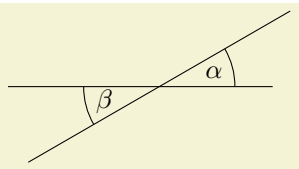
7.9 The Arc Operation

The *arc operation* allows you to add an arc to the current path. You provide a start angle, an end angle, and a radius. The arc operation will then add a part of a circle of the given radius between the given angles. The arc will start at the current point and will end at the end of the arc.

Syntax for the arc operation: `arc(`⟨*start angle*⟩`:`⟨*end angle*⟩`:`⟨*radius*⟩`/`⟨*half height*⟩`)`.



```
\begin{tikzpicture}
  \draw (0,0) arc (180:90:1cm) -- (2,.5) arc (90:0:1cm);
  \draw (4,0) -- +(30:1cm) arc (30:60:1cm) -- cycle;
  \draw (8,0) arc (0:270:1cm/.5cm) -- cycle;
\end{tikzpicture}
```



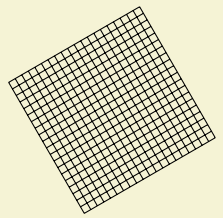

```
\begin{tikzpicture}
  \draw (-1,0) -- +(3.5,0);
  \draw (1,0) ++(210:2cm) -- +(30:4cm);
  \draw (1,0) +(0:1cm) arc (0:30:1cm);
  \draw (1,0) +(180:1cm) arc (180:210:1cm);
  \path (1,0) ++(15:.75cm) node{$\alpha$};
  \path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

7.10 The Grid Operation

You can add a grid to the current path using the `grid` path operation.

Syntax for the grid operation: `grid`⟨*corner coordinate*⟩.

The effect of the grid operation is the following: It will draw a grid filling a rectangle whose two corners are given by ⟨*corner coordinate*⟩ and by the previous coordinate. Thus, the typical way in which a grid is drawn is `\draw (1,1) grid (3,3);`, which yields a grid filling the rectangle whose corners are at $(1, 1)$ and $(3, 3)$. All coordinate transformations apply to the grid.



```
\tikz[rotate=30] \draw[step=1mm] (0,0) grid (2,2);
```

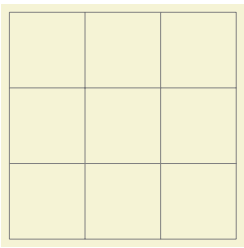
The stepping of the grid is governed by the following options:

- `step`=⟨*dimension*⟩ sets the stepping in both the x and y-direction.
- `xstep`=⟨*dimesion*⟩ sets the stepping in the x-direction.
- `ystep`=⟨*dimesion*⟩ sets the stepping in the y-direction.

It is important to note that the grid is always "phased" such that it contains the point $(0, 0)$ if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that due to rounding errors, the "last" lines of a grid may be ommitted. In this case, you have to add an epsilon to the corner points.

The following styles are useful for drawing grids:

- `style=help lines` This style makes lines "subdued"by using thin gray lines for them. However, this style is not installed automatically and you have to say for example:

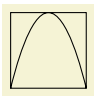


```
\tikz \draw[style=help lines] (0,0) grid (3,3);
```

7.11 The Parabola Operation

The `parabola` path command continues the current path with a parabola. A parabola is a (shifted and scaled) curve defined by the equation $f(x) = x^2$ and looks like this: $\cup$.

The basic syntax of the parabola command is `parabola`⟨*coordinate*⟩. This will draw a parabola that "fills" the rectangle whose corners are the old current point and ⟨*coordinate*⟩. Here is an example:



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

In detail, the parabola has the following properties: The old current point lies in one corner of the rectangle and the parabola "goes through" this point. The parabola will *not* end at the given ⟨*coordinate*⟩. Rather, it will have its bend on the y-coordinate of ⟨*coordinate*⟩. It will end on the x-coordinate of ⟨*coordinate*⟩ and the y-coordinate of the old current point.

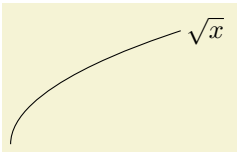
Sometimes you may want to draw only a "part" of a parabola. For this, you can use `parabola left` and `parabola right`.

The syntax of the first command is `parabola left`⟨*coordinate*⟩. This will draw the "left part" of a parabola. More precisely, a parabola is drawn that goes through the cold current point and has its bend at ⟨*coordinate*⟩. Here is an example: \, which is obtained through the command `\tikz \draw (-1ex,1.5ex) parabola left (0,0);`.

Similarly, there exists a `parabola right`⟨*coordinate*⟩ command. Here, the parabola has its bend at the old current point and goes through the new ⟨*coordinate*⟩.

You can use `parabola left` and `parabola right` to draw parts of parabolas. However, it is not possible to draw only a part of a parabola that does not contain the bend. For this you need to use a `plot` command.

```
\tikz \draw[scale=0.25] (-1,1) parabola left (0,0) parabola right (2,4);
```



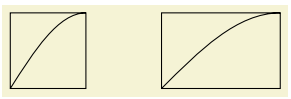

```
\tikz \draw[rotate=-90] (-1.5,2.25) node[right]{$\sqrt x$} parabola left(0,0);
```

7.12 The Sine and Cosine Operation

The `sin` and `cos` operations are similar to the `parabola` command. They, too, can be used to draw (parts of) a sine or cosine curve.

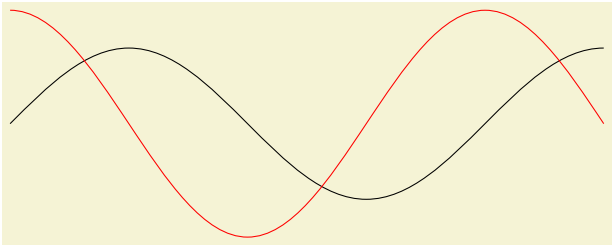
Syntax: `sin`⟨*coordiante*⟩ and `cos`⟨*coordinate*⟩.

The effect of `sin` is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and that the end of the curve in the interval is at ⟨*coordinate*⟩. Here is an example that should clarify this:



```
\tikz \draw (0,0) rectangle (1,1)     (0,0) sin (1,1)
          (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

The `cos` operation works similarly, only a cosine in the interval $[0, \pi/2]$ is drawn. By correctly alternating `sin` and `cos` operations, you can create a complete sine or cosine curve:



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
\end{tikzpicture}
```

Note that there is no way to (conveniently) draw an interval on a sine or cosine curve whose end points are not multiples of $\pi/2$ (or 90°).

7.13 The Plot Operation

The `plot` operation can be used to append a line or curve to the path that goes through a large numbor of coordiantes. These coordinates are either given in a simple list of coordinates or they are read from some file.

The syntax of the `plot` comes in different versions. First, the "beginning" of a `plot` command can be

1. `plot` or
2. `--plot` (space may be added for clarity)

The difference between the first and the second case is that `plot` will start plotting at the first coordinate by "moving" to that point. The `--plot` will instead perform a `--` (line-to) operation to the first point of the plot.

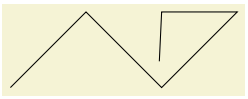
Next, there are three ways of specifying the coordinates of the points to be plotted:

1. `--plot[`⟨*local options*⟩`](`⟨*coordinate 1*⟩⟨*coordinate 2*⟩...⟨*coordinate n*⟩`)`
2. `--plot[`⟨*local options*⟩`]"`⟨*filename*⟩`"`
3. `--plot[`⟨*local options*⟩`]$`⟨*gnuplot formula*⟩`$`

These different ways are explained in the following.

7.13.1 Plotting Points Given Inline

In the first two cases, the points are given directly in the TEX-file as in the following example:



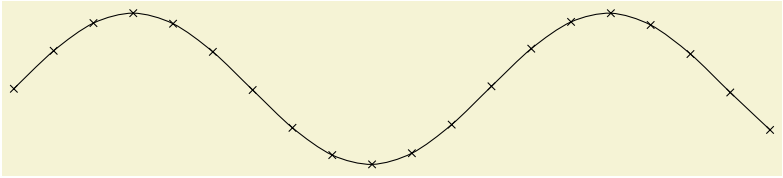
```
\tikz \draw plot ((0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm));
```

Here is an example showing the difference between `plot` and `--plot`:

```
\begin{tikzpicture}
  \draw (0,0) -- (1,1) plot ((2,0)  (4,0));
  \draw[color=red,xshift=5cm]
        (0,0) -- (1,1) -- plot ((2,0)  (4,0));
\end{tikzpicture}
```

7.13.2 Plotting Points Read From an External File

In the third and fourth form the points reside in an external file named ⟨*filename*⟩. Currently, the only file format that Ti*k*Z allows is the following: Each line of the ⟨*filename*⟩ should contain one line starting with two numbers, separated by a space. Anything following the two numbers on the line is ignored. Also, lines starting with a `%` or a `#` are ignored as well as empty lines. (This is exactly the format that GNUPLOT produces when you say `set terminal table`.) If necessary, more formats will be supported in the future, but it is usually easy to produce a file containing data in this form.



```
\tikz \draw plot[mark=x,smooth] "plots/pgfmanual-sine.table";
```

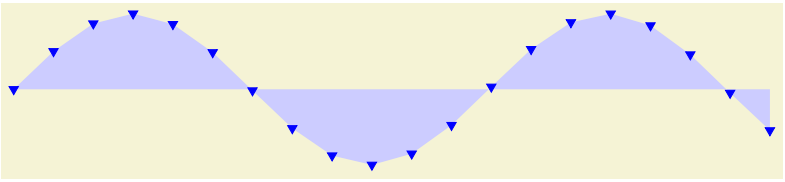
The file `plots/pgfmanual-sine.table` reads:

```
#Curve 0, 20 points
#x y type
0.00000 0.00000  i
0.52632 0.50235  i
1.05263 0.86873  i
1.57895 0.99997  i
...
9.47368 -0.04889  i
10.00000 -0.54402  i
```

It was produced from the following source, using `gnuplot`:

```
set terminal table
set output "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
plot [x=0:10] sin(x)
```

The ⟨*local options*⟩ of the `plot` command are local to each plot and do not affect any other plots "on the same path." Most importantly, they affect how marks are drawn on the plot, but graphic state options and transformations given in the ⟨*local options*⟩ *do not affect the plot itself.* This means, that when you say `\tikz \draw plot[marks=x,color=red] ((0,0) (1,1) (2,0));`, the plot marks will be red, whereas the plot will be black (or whatever drawing color was in force). Similarly, a local option `rotate=180` will turn ticks upside down, but will not affect the plot path. An option given *outside* will, however, affect the path. Here is an example:



```
\tikz \fill[fill=blue!20,mark=triangle*]
  plot[color=blue,rotate=180] "plots/pgfmanual-sine.table" |- (0,0);
```

7.13.3 Plotting a Function

Often, you will want to plot points that are given via a function like $f(x) = x \sin x$. Unfortunately, TEX does not really have enough computational power to generate the points on such a function efficiently (it is a text processing program, after all). However, if you allow it, TEX can try to call external programs that can easily produce the necessary points. Currently, PGF knows how to call GNUPLOT.

When Ti*k*Z encounters your command `plot[id=`⟨*id*⟩`] $x*sin(x)$` for the first time, it will create a file called ⟨*prefix*⟩⟨*id*⟩`.gnuplot`, where ⟨*prefix*⟩ is `\jobname.` by default, that is, the name of you main `.tex` file. If no ⟨*id*⟩ is given, it will be empty, which is allright, but it is better when each plot has a unique ⟨*id*⟩ for reasons explained in a moment. Next, Ti*k*Z writes some initialization code into this file followed by `plot x*sin(x)`. The initialization code sets up things such that the `plot` command will write the coordinates into another file called ⟨*prefix*⟩⟨*id*⟩`.table`. Finally, this table file is read as if you had said `plot "`⟨*prefix*⟩⟨*id*⟩`.table"`.

For the plotting mechansim to work, two conditions must be met:

1. You must have allowed TEX to call external programs. This is often switched off by default since this is a securtiy risk (you might, without knowing, run a TEX file that calls all sorts of "bad" commands). To enable this "calling external programs" a command line option must be given to the TEX program. Usually, it is called something like `shell-escape` or `enable-write18`. For example, for my `pdflatex` the option `--shell-escape` can be given.

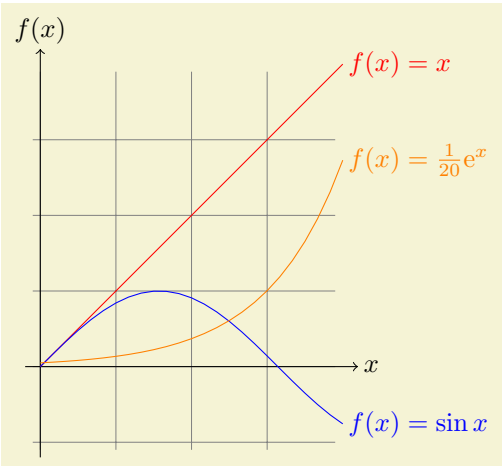
2. You must have installed the `gnuplot` program and TEX must find it when compiling your file.

Unfortunately, these conditions will not always be met. Especially if you pass some source to a coauthor and the coauthor does not have GNUPLOT installed, he or she will have trouble compiling your files.

For this reason, Ti*k*Z behaves differently when you compile your graphic for the second time: If upon reaching `plot[id=`⟨*id*⟩`] $...$` the file ⟨*prefix*⟩⟨*id*⟩`.table` already exists *and* if the ⟨*prefix*⟩⟨*id*⟩`.gnuplot` file contains what Ti*k*Z thinks that it "should" contain, the `.table` file is immediately read without trying to call a `gnuplot` program. This approach has the following advantages:

1. If you pass a bundle of your `.tex` file and all `.gnuplot` and `.table` files to someone else, that person can TEX the `.tex` file without having to have `gnuplot` installed.
2. If the `\write18` feature is switched off for security reasons (a good idea), then, upon the first compilation of the `.tex` file, the `.gnuplot` will still be generated, but not the `.table` file. You can then simply call `gnuplot` "by hand" for each `.gnuplot` file, which will produce all necessary `.table` files.
3. If you change the function that you wish to plot or its domain, Ti*k*Z will automatically try to regenerate the `.table` file.
4. If, out of laziness, you do not provide an `id`, the same `.gnuplot` will be used for different plots, but this is not a problem since the `.table` will automatically be regenerated for each plot on-the-fly. *Note: If you intend to share your files with someone else, always use an id, so that the file can by typset without having* GNUPLOT *installed.* Also, having unique ids for each plot will improve compilation speed since no external programs need to be called, unless it is really necessary.

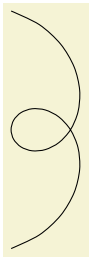
When you use `plot $`⟨*gnuplot formula*⟩`$`, the ⟨*gnuplot formula*⟩ must be given in the `gnuplot` syntax, whose details are beyond the scope of this manual. Here is the ultra-condensed essence: Use `x` as the variable and use the C-syntax for normal plots, use `t` as the variable for parametric plots. Here are some examples:




```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

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

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

The following options influence the plot:

- `samples=`⟨*number*⟩ sets the number of samples used in the plot. The default is 25.
- `domain=`⟨*start*⟩`:`⟨*end*⟩ sets the domain between which the samples are taken. The default is `-5:5`.
- `parametric=`⟨*true or false*⟩ sets whether the plot is a parameteric plot. If true, then `t` must be used instead of `x` as the parameter and two comma-separated functions must be given in the ⟨*gnuplot formula*⟩. An example is the following:



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
     plot[parametric,id=parametric-example] $t*sin(t),t*cos(t)$;
```

- `id`=⟨*id*⟩ sets the identifier of the current plot. This should be a unique identifier for each plot (though things will also work if it is not, but not as well, see the explanations above). The ⟨*id*⟩ will be part of a filename, so it should not contain anything fancy like `*` or `$`.
- `prefix`=⟨*prefix*⟩ is put before each plot file name. The default is `\jobname.`, but if you have many plots, it might be better to use, say `plots/` and have all plots placed in a directory. You have to create the director yourself.
- `raw gnuplot` causes the ⟨*gnuplot formula*⟩ to be passed on to GNUPLOT without setting up the samples or the `plot` command. Thus, you could write `plot[raw gnuplot,id=raw-example] $set samples 25; plot sin(x)$`. This can be useful for complicated things that need to be passed to GNUPLOT. However, for really complicated situations you should create a special external generating GNUPLOT file and use the `"`-syntax to include the table "by hand."

The following styles influence the plot:

- `style=plot` This style is installed in each plot, that is, as if you always said `plot[style=plot,...]`. This is most useful for globally setting a prefix for all plots by saying:

```
\tikzstyle plot=[prefix=plots/]
```

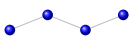
7.13.4 Placing Marks on the Plot

As we saw already, it is possible to add *marks* to a plot using the `mark` option. When this option is used, a copy of the plot mark is placed on each point of the plot. Note that the marks are placed *after* the whole path has been drawn/filled/shaded. In this respect, they are handled like text nodes. Any options given as ⟨*local options*⟩ to the `plot` command affect how the marks are drawn.

In detail, the following options govern how marks are drawn:

- `mark`=⟨*mark mnemonic*⟩ Sets the mark to a mnemonic that has previously been defined using the `\newpgfplotmark`. By default, `*`, `+`, and `x` are available, which draw a filled circle, a plus, and a cross as marks. Many more marks become available when the library `pgflibraryplotmarks` is loaded. Section 11.2.3 lists the available plot marks.

  One plot mark is special: the `ball` plot mark is available only it Ti*k*Z. The `ball color` determines the balls's color. Do not use this option with large number of marks since it will take very long to render in PostScript.

Option	Effect
—mark=—ball	

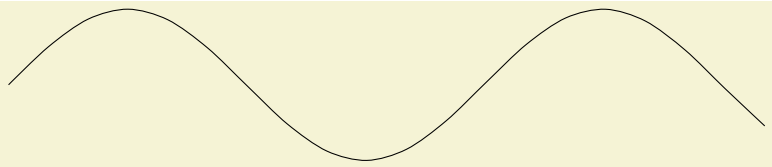
- `mark size`=⟨*dimension*⟩ Sets the size of the plot marks. For circular plot marks, ⟨*dimension*⟩ is the radius, for other plot marks ⟨*dimension*⟩ should be about half the width and height.

  This option is not really necessary, since you achieve the same effect by saying `scale`=⟨*factor*⟩, where ⟨*factor*⟩ is the quotient of the desired size and the default size. However, using `mark size` is a bit faster and somehow more natural.

7.13.5 Smooth Plots, Sharp Plots, and Comb Plots

There are different things the `plot` command can do with the points it reads from a file or from the inlined list of points. By default, it will connect these points by straight lines. However, you can also use options to change the behaviour of `plot`.

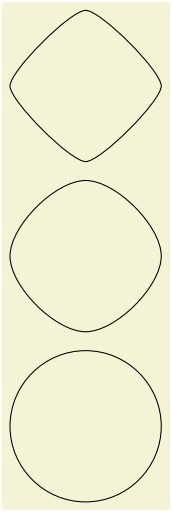
- `sharp plot` This is the default and causes the points to be connected by straight lines. This option is included only so that you can "switch back" if you "globally" install, say, `smooth`.
- `smooth` This option causes the points on the path to be connected using a smooth curve:



```
\tikz\draw plot[smooth] "plots/pgfmanual-sine.table";
```

Note that the smoothing algorithm is not very intelligent. You will get the best results if the bending angles are small, that is, less than about 30° and, even more importantly, if the distances between points are about the same all over the plotting path.

- `tension`=⟨*value*⟩ This option influences how "tight" the smoothing is. A lower value will result in sharper corners, a higher value in more "round" curves. The default is 0.15. The "correct" value depends on the details of plot.

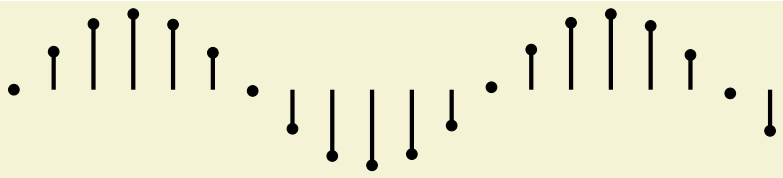


```
\begin{tikzpicture}[smooth cycle]
  \draw                 plot[tension=0.1]   ((0,0) (1,1) (2,0) (1,-1));
  \draw[yshift=-2.25cm] plot[tension=0.2]   ((0,0) (1,1) (2,0) (1,-1));
  \draw[yshift=-4.5cm]  plot[tension=0.275] ((0,0) (1,1) (2,0) (1,-1));
\end{tikzpicture}
```

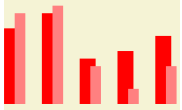
- `smooth cycle` This option causes the points on the path to be connected using a closed smooth curve.

```
\tikz[scale=0.5] \draw plot[smooth cycle] ((0,0) (1,0) (2,1) (1,2))
                       plot               ((0,0) (1,0) (2,1) (1,2)) -- cycle;
```

- `ycomb` This option causes the `plot` command to interpret the plotting points differently. Instead of connecting them, for each point of the plot a straight line is added to the path from the x-axis to the point, resulting in a sort of "comb" or "bar diagram."



```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] "plots/pgfmanual-sine.table";
```

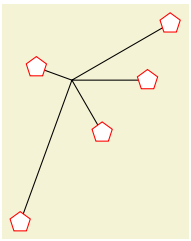


```
\begin{tikzpicture}[ycomb]
  \draw[color=red,line width=6pt]
    plot ((0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9));
  \draw[color=red!50,line width=4pt,xshift=3pt]
    plot ((0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5));
\end{tikzpicture}
```

- `xcomb` This option works like `ycomb` except that the bars are horizontal.

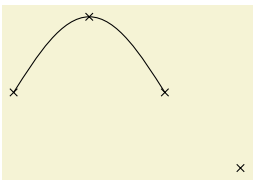
```
\tikz \draw plot[xcomb,mark=x] ((1,0) (0.8,0.2) (0.6,0.4) (0.2,1));
```

- `polar comb` This option causes a line from the origin to the point to be added to the path for each plot point.



```
\tikz \draw plot[polar comb,mark=pentagon*,fill=white,draw=red,mark size=4pt]
    ((0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm));
```

- `only marks` This option causes only marks to be shown; no path segments are added to the actual path. This can be useful for quickly adding some marks to a path.



```
\tikz \draw (0,0) sin (1,1) cos (2,0)
          plot[only marks,mark=x] ((0,0) (1,1) (2,0) (3,-1));
```

7.14 The Scoping Operation

When Ti*k*Z encounters and opening or a closing brace (`{` or `}`) at some point where a path operation should come, it will open or close a scope. All options that can be applied "locally" will be scoped inside the scope. For example, if you apply a transformation like `[xshift=1cm]` inside the scoped area, the shifting only applies to the scope. On the other hand, a command like `color=red` does not have any effect inside a scope since it can only be applied to the path as a whole.

7.15 The Node Operation

You can add nodes to a path using the `node` operation. Since this operation is quite complex and since the nodes are not really part of the path itself, there is a separate section dealing with nodes, see Section 9.

8 Actions on Paths

Once a path has been constructed, different things can be done with it. It can be drawn (or stroked) with a "pen," it can be filled with a color or shading, it can be used for clipping subsequent drawing, it can be used to specify the extend of the picture—or all or any combination of these actions at the same time.

To decide what is to be done with a path, two methods can be used. First, you can use a special-purpose command like `\draw` to indicate that the path should be drawn. However, commands like `\draw` and `\fill` are just abbreviations for special cases of the more general method: Here, the `\path` command is used to specify the path. Then, options encountered on the path indicate what should be done with the path.

For example, `\path (0,0) circle (1cm);` means "This is a path consisting of a circle around the origin. Do not do anything with it (throw it away)." However, if the option `draw` is encountered anywhere on the path, the circle will be drawn. "Anywhere" is any point on the path where an option can be given, which is everywhere where a path command like `circle (1cm)` or `rectangle (1,1)` or even just `(0,0)` would also be allowed. Thus, the following commands all draw the same circle:

```
\path [draw] (0,0) circle (1cm);
\path (0,0) [draw] circle (1cm);
\path (0,0) circle (1cm) [draw];
```

Finally, `\draw (0,0) circle (1cm);` also draws a path, because `\draw` is an abbreviation for `\path [draw]` and thus the command expands to the first line of the above example.

Similarly, `\fill` is an abbreviation for `\path[fill]` and `\filldraw` is an abbreviation for the command `\path[fill,draw]`. Since options accumulate, the following commands all have the same effect:

```
\path [draw,fill]   (0,0) circle (1cm);
\path [draw] [fill] (0,0) circle (1cm);
\path [fill] (0,0) circle (1cm) [draw];
\draw [fill] (0,0) circle (1cm);
\fill (0,0) [draw] circle (1cm);
\filldraw (0,0) circle (1cm);
```

In the following subsection the different actions are explained that can be used with a path. The following commands are abbreviations for certain sets of actions, but for many useful combinations there are no abbreviations:

`\draw`

Inside `{tikzpicture}` this is an abbreviation for `\path[draw]`.

`\fill`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill]`.

`\filldraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill,draw]`.

`\shade`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade]`.

`\shadedraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade,draw]`.

`\clip`

Inside `{tikzpicture}` this is an abbreviation for `\path[clip]`.

`\useasboundingbox`

Inside `{tikzpicture}` this is an abbreviation for `\path[use as bounding box]`.

`\node`

Inside `{tikzpicture}` this is an abbreviation for `\path node`. Note that, for once, `node` is not an option but a path operation.

`\coordinate`

Inside `{tikzpicture}` this is an abbreviation for `\path coordinate`.

8.1 Specifying Colors

The most unspecific option for setting colors is the following:

- `color=`⟨*color name*⟩ This option sets the color that is used for fill, drawing, and text inside the current scope. Any special settings for filling colors or drawing colors are immediately "overruled" by this option.

  The ⟨*color name*⟩ is the name of a previously defined color. For LaTeX users, this is just a normla "LaTeX-color" and the `xcolor` extensions are allows. Here is an example:


  ```
  \tikz \fill[color=red!20] (0,0) circle (1ex);
  ```

  It is possible to "leave out" the `color=` part if you load the `xkeyval` package. Thus, if this package is loaded, you can also write


  ```
  \tikz \fill[red!20] (0,0) circle (1ex);
  ```

  What happens is the every option that Ti*k*Z does not know, like `red!20` gets a "second chance" as a color name.

  For plain TeX users, it is not so easy to specify colors since plain TeX has no "standarderized" color naming mechanism. Because of this, PGF emulates the `xcolor` package, though the emulation is *extremely basic* (more precisely, what I could hack together in two hours or so). The emulation allows you to do the following:

  - Specify a new color using `\definecolor`. Only the two color models `gray` and `rgb` are supported.

    Example: `\definecolor{orange}{rgb}{1,0.5,0}`
  - Use `\colorlet` to define a new color based on an old one. Here, the `!` mechanism is supported, though only "once" (use multiple `\colorlet` for more fancy colors).

    Example: `\colorlet{lightgray}{black!25}`
  - Use `\color{`⟨*color name*⟩`}` to set the color in the current TeX group. `\aftergroup`-hackery is used to restore the color after the group.

As pointed out above, the `color=` option applies to "everything" (except to shadings), which is not always what you want. Because of this, there are several more specialised color options. For example, the `draw=` option sets the color used for drawing, but does not modify the color used for filling. These color options are documented where the path action they influence is desribed.

8.2 Drawing a Path

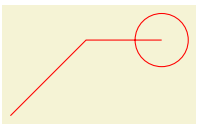
You can draw a path using the following option:

- `draw=`⟨*color*⟩ Causes the path to be drawn. "Drawing" (also known as "stroking") can be thought of as picking up a pen and moving it along the path, thereby leaving "ink" on the canvas.

  There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

  If the optional ⟨*color*⟩ argument is given, drawing is done using the given ⟨*color*⟩. This color can be different from the current filling color, which allows you to draw and fill a path with different colors. If no ⟨*color*⟩ argument is given, the last usage of the `color=` option is used.

  Although this option is normally used on paths to indicate that the path should be drawn, it also makes sense to use the option with a `{scope}` or `{tikzpicture}` environment. However, this will *not* cause all path to drawn. Instead, this just sets the ⟨*color*⟩ to be used for drawing paths inside the environment.

  ```
  \begin{tikzpicture}
    \path[draw=red] (0,0) -- (1,1) -- (2,1) circle (10pt);
  \end{tikzpicture}
  ```

The following subsections list the different options that influence how a path is drawn. All of these options only have an effect if the `draw` options is given (directly or indirectly).

8.2.1 Line Width, Line Cap, and Line Join Options

- `line width`=⟨*dimension*⟩ Specifies the line width. Note the space. Default: `0.4pt`.

```
\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);
```

There are a number of predefined styles that provide more "natural" ways of setting the line width. You can also redefine these styles. Remember that you can leave out the `style=` when setting a style.

- `style=ultra thin` Sets the line width to 0.1pt.

```
\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);
```

- `style=very thin` Sets the line width to 0.2pt.

```
\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);
```

- `style=thin` Sets the line width to 0.4pt.

```
\tikz \draw[thin] (0,0) -- (1cm,1.5ex);
```

- `style=semithick` Sets the line width to 0.6pt.

```
\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);
```

- `style=thick` Sets the line width to 0.8pt.

```
\tikz \draw[thick] (0,0) -- (1cm,1.5ex);
```

- `style=very thick` Sets the line width to 1.2pt.

```
\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);
```

- `style=ultra thick` Sets the line width to 1.6pt.

```
\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);
```

- `cap`=⟨*type*⟩ Specifies how lines "end." Permissble ⟨*type*⟩ are `round`, `rect`, and `butt` (default). They have the following effects:

```
\begin{tikzpicture}
  \begin{scope}[line width=10pt]
    \draw[cap=rect] (0,0 ) -- (1,0);
    \draw[cap=butt] (0,.5) -- (1,.5);
    \draw[cap=round] (0,1 ) -- (1,1);
  \end{scope}
  \draw[white,line width=1pt]
    (0,0 ) -- (1,0) (0,.5) -- (1,.5) (0,1 ) -- (1,1);
\end{tikzpicture}
```

- `join`=⟨*type*⟩ Specifies how lines "join." Permissble ⟨*type*⟩ are `round`, `bevel`, and `miter` (default). They have the following effects:

```
\begin{tikzpicture}[line width=10pt]
  \draw[join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
\end{tikzpicture}
```

- `miter limit`=⟨*factor*⟩ When you use the miter join and there is a very sharp corner (a small angle), the miter join may protrude very far over the actual joining point. In this case, if it were to protrude by more than ⟨*factor*⟩ times the line width, the miter join is replaced by a bevel join. Deault value is `10`.

```
\begin{tikzpicture}[line width=5pt]
  \draw                 (0,0) -- ++(5,.5) -- ++(-5,.5);
  \draw[miter limit=25] (6,0) -- ++(5,.5) -- ++(-5,.5);
\end{tikzpicture}
```

8.2.2 Dash Patterns

- `dash pattern`=⟨*dash pattern*⟩ Sets the dashing pattern. The syntax is the same as in METAFONT. For example `on 2pt off 3pt on 4pt off 4pt` means "draw 2pt, then leave out 3pt, then draw 4pt once more, then leave out 4pt again, repeat".


```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
  \draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

- `dash phase`=⟨*dash phase*⟩ Shifts the start of the dash pattern by ⟨*phase*⟩.


```
\begin{tikzpicture}[dash pattern=on 20pt off 10pt]
  \draw[dash phase=0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw[dash phase=10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

As for the line thickness, some predefined styles allow you to set the dashing conveniently.

- `style=solid` Shorthand for setting a solid line as "dash pattern." This is the default.


```
\tikz \draw[solid] (0pt,0pt) -- (50pt,0pt);
```

- `style=dotted` Shorthand for setting a dotted dash pattern.


```
\tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=densely dotted` Shorthand for setting a densely dotted dash pattern.


```
\tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=loosely dotted` Shorthand for setting a loosely dotted dash pattern.


```
\tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=dashed` Shorthand for setting a dashed dash pattern.


```
\tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt);
```

- `style=densely dashed` Shorthand for setting a densely dashed dash pattern.


```
\tikz \draw[densely dashed] (0pt,0pt) -- (50pt,0pt);
```

- `style=loosely dashed` Shorthand for setting a loosely dashed dash pattern.


```
\tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt);
```

8.2.3 Arrows

When you draw a line, you can add arrows at the ends. Currently, it is only possible to add one arrow at the start and one at the end. Thus, even if the path consists of several segments, only the first and last segments get arrows. In general, it is a good idea to add arrows only to paths that consist of a single, unbroken line. The behaviour for paths that consist of several segments is not specified and may change in the future.

- `arrows=`⟨*start arrow kind*⟩`-`⟨*end arrow kind*⟩ This option sets the start and end arrows (an empty value as in `->` indicates that no arrow should be drawn at the start).

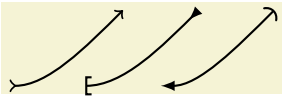
  Note: Since the arrow option is so often used, you can leave out the text `arrows=`. What happens is that every option that contains a `-` is interpreted as an arrow specification.

```
\begin{tikzpicture}
  \draw[->]        (0,0)   -- (1,0);
  \draw[o-stealth] (0,0.1) -- (1,0.1);
\end{tikzpicture}
```

  The permissible values are all defined arrows, though you can also define new arrow kinds as explained in Section 20. This is often necessary to obtain "double" arrows and arrows that have a fixed size. Since `pgflibraryarrows` is loaded by default, all arrows described in Section 11.1 are available.

  One arrow kind is special: `>` (and all arrow kinds containing the arrow king such as `<<` or `>|`). This arrow type is not fixed. Rather, you can redefine it using the `>=` option, see below.

  Example: You can also combine arrow types as in



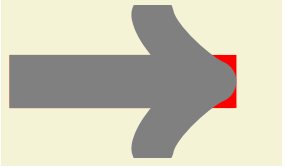
```
\begin{tikzpicture}[thick]
  \draw[to reversed-to]   (0,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[[-latex reversed] (1,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[latex-)]          (2,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
\end{tikzpicture}
```

- `>=`⟨*end arrow kind*⟩ This option can be used to redefine the "standard" arrow `>`. The idea is that different people have different ideas what arrow kind should normally be used. I prefer the arrow of TEX's `\to` command (which is used in things like $f\colon A \to B$). Other people will prefer LATEX's standard arrow, which looks like this: $\rightarrow$. Since the arrow kind `>` is certainly the most "natural" one to use, it is kept free of any predefined meaning. Instead, you can change it by saying `>=to` to set the "standard" arrow kind to TEX's arrow, whereas `>=latex` will set it to LATEX's arrow and `>=stealth` will use a PSTRICKS-like arrow.

  Apart from redefining the arrow kind `>` (and `<` for the start), this option also redefines the following arrow kinds: `>` and `<` as the swapped version of ⟨*end arrow kind*⟩, `<<` and `>>` as doubled versions, `>>` and `<<` as swapped doubled versions, and `|<` and `>|` as arrows ending with a vertical bar.

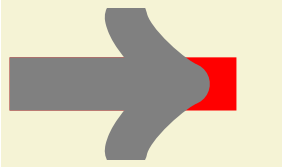
```
\begin{tikzpicture}
  \begin{scope}[>=latex]
    \draw[->]    (0pt,6ex) -- (1cm,6ex);
    \draw[>->>]  (0pt,5ex) -- (1cm,5ex);
    \draw[|<->|] (0pt,4ex) -- (1cm,4ex);
  \end{scope}
  \begin{scope}[>=diamond]
    \draw[->]    (0pt,2ex) -- (1cm,2ex);
    \draw[>->>]  (0pt,1ex) -- (1cm,1ex);
    \draw[|<->|] (0pt,0ex) -- (1cm,0ex);
  \end{scope}
\end{tikzpicture}
```

- `shorten >=`⟨*dimension*⟩ This option will shorten the end of lines by the given ⟨*dimension*⟩. If you specify an arrow, lines are already shortened a bit such that the arrow touches the specified endpoint and does not "protrude over" this point. Here is an example:



```
\begin{tikzpicture}[line width=20pt]
  \clip (0,0) rectangle (3.5,2);
  \draw[red]     (0,1) -- (3,1);
  \draw[gray,->] (0,1) -- (3,1);
\end{tikzpicture}
```

The `shorten >` option allows you to shorten the end on the line *additionally* by the given distance. This option can also be useful if you have not specified an arrow at all.

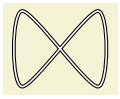


```
\begin{tikzpicture}[line width=20pt]
  \clip (0,0) rectangle (3.5,2);
  \draw[red]                    (0,1) -- (3,1);
  \draw[-to,shorten >=10pt,gray] (0,1) -- (3,1);
\end{tikzpicture}
```

- `shorten <`=⟨*dimension*⟩ works like `shorten >`.

8.2.4 Double Lines and Border Lines

- `double`=⟨*core color*⟩ This option causes "two" lines to be drawn instead of a single one. However, this is not what really happens. In reality, the path is drawn twice. First, with the normal drawing color, secondly with the ⟨*core color*⟩, which is normally `white`. Upon the second drawing, the line width is reduced. The net effect is that it appears as if two lines had been drawn and this works well even with complicated, curved paths:



```
\tikz \draw[double] plot[smooth cycle] ((0,0) (1,1) (1,0) (0,1));
```

You can also use the doubling option to create an effect in which a line seems to have a certain "border":

```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

- `double distance`=⟨*dimension*⟩ Sets the distance the "two" are spaced apart (default is 0.6pt). In reality, this is the thickness of the line that is used to draw the path for the second time. The thickness of the *first* time the path is drawn is twice the normal line width plus the given ⟨*dimension*⟩. As a side-effect, this option "selects" the `double` option.

```
\begin{tikzpicture}
  \draw[very thick,double]              (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt] (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt]       (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

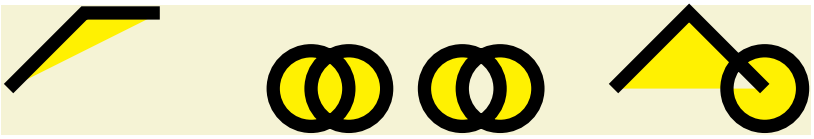
8.3 Filling a Path

To fill a path, you use the following option:

- `fill`=⟨*color*⟩ This option causes the path to be filled. All unclosed parts of the path are first closed, if necessary. Then, the area enclosed by the path is filled with the current filling color, which is either the last color set using the general `color=` option or the optional color ⟨*color*⟩. For self-intersection paths and for paths consisting of several closed areas, the "enclosed area" is somewhat complicated to define and two different definitions exist, namely the nonzero winding number rule and the even odd rule, see the explanation of these options, below.

```
\begin{tikzpicture}
  \fill (0,0) -- (1,1) -- (2,1);
  \fill (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \fill[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \fill (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

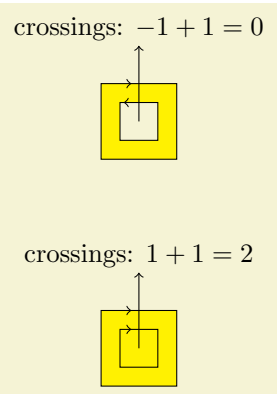
If the `fill` option is used together with the `draw` option (either because both are given as options or because a `\filldraw` command is used), the command is draw *firstly*, then the path is filled *secondly*. This is especially useful if different colors are selected for drawing and for filling. Even if the same color is used, there is a difference between this command and a plain `fill`: A "filldrawn" area will be slightly larger than a filled area because of the thickness of the "pen."



```
\begin{tikzpicture}[fill=yellow,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \filldraw[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

The following two options can be used to decide on which filling rule should be used:

- `nonzero rule` If this rule is used (which is the default), the following method is used to determine whether a given point is "inside" the path: From the point, shoot a ray in some direction towards infinity (the direction is chosen such that no strange borderline cases occur). Then the ray may hit the path. Whenever it hits the path, we increase or decrease a counter, which is initially zero. If the ray hits the path as the path goes "from left to right" (relative to the ray), the counter is increased, otherwise it is decreased. Then, at the end, we check whether the counter is nonzero (hence the name). If so, the point is deemed to lie "inside," otherwise it is "outside." Sounds complicated? It is.

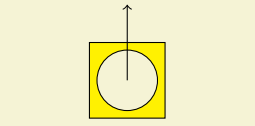



```
\begin{tikzpicture}
  \filldraw[fill=yellow]
  % Clockwise rectangle
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % Counter-clockwise rectangle
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

  \draw[->] (0,1) (.4,1);
  \draw[->] (0.75,0.75) (0.3,.75);

  \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};

  \begin{scope}[yshift=-3cm]
    \filldraw[fill=yellow]
    % Clockwise rectangle
    (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
    % Clockwise rectangle
    (0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;

    \draw[->] (0,1) (.4,1);
    \draw[->] (0.25,0.75) (0.4,.75);

    \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $1+1 = 2$};
  \end{scope}
\end{tikzpicture}
```

- `even odd rule` This option causes a different method to be used for determining the inside and outside of paths. Will it is less flexible, it turns out to be more intuitive.

  With this method, we also shoot rays from the point for which we wish to determine wheter it is inside or outside the filling area. However, this time we only count how often we "hit" the path and declare the point to be "inside" if the number of hits is odd.

  Usin the even-odd rule, it is easy to "drill holes" into a path.




```
\begin{tikzpicture}
  \filldraw[fill=yellow,even odd rule]
    (0,0) rectangle (1,1) (0.5,0.5) circle (0.4cm);
  \draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
\end{tikzpicture}
```

8.4 Shading a Path

You can shade a path using the `shade` option. A shading is like a filling, only the shading changes its color smoothly from one color to another.

- `shade` Causes the path to be shaded using the currently selected shading (more on this later). If this option is used together with the `draw` option, then the path is first shaded, then drawn.

  It is not an error to use this option together with the `fill` option, but it makes no sense.


```
\tikz \shade (0,0) circle (1ex);
```

```
\tikz \shadedraw (0,0) circle (1ex);
```

For some shadings it is not really clear how they can "fill" the path. For example, the `ball` shading normally looks like this:  . How is this supposed to fill a rectangle? Or a triangle?

To solve this problem, the predefined shadings like `ball` or `axis` fill a large rectangle completely in a sensible way. Then, when the shading is used to "fill" a path, what actually happens is that the path is temporarily used for clipping and then the rectangular shading is drawn, scaled and shifted such that all parts of the path are filled.

8.4.1 Choosing a Shading Type

As can be seen, the default shading is a smooth transition from gray to white and from above to bottom. However, other shadings are also possible, for example a shading that will sweep a color from the center to the corners outward. To choose the shading, you can use the `shading=` option which will also automatically invoke the `shade` option. Note that this does *not* change the shading color, only the way the colors sweep. For changing the colors, other options are needed, which are explained below.

- `shading`=⟨*name*⟩ This selects a shading named ⟨*name*⟩. The following shadings are predefined:
  - `axis` This is the default shading in which the color changes gradually between two three horizontal lines. The top line is at the top (uppermost) point of the path, the middle is in the middle, the bottom line is at the bottom of the path.

```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
```

    The default top color is gray, the default bottom color is white, the default middle is the "middle" of these two.
  - `radial` This shading fills the path with a gradual sweep from a certain color in the middle to another color at the border. If the path is a circle, the outer color will be reached exactly at the border. If the shading is not a circle, the outer color will continue a bit towards the corners. The default inner color is gray, the deafult outer color is white.

```
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
```

  - `ball` This shading fills the path with a shading that "looks like a ball." The default "color" of the ball is blue (for no particular reason).

```
\tikz \shadedraw [shading=ball] (0,0) rectangle (1,1);
```

```
\tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
```

- `shading angle`=⟨*degrees*⟩ This option rotates the shading (not the path!) by the given angle. For example, we can turn a top-to-bottom axis shading into a left-to-right shading by rotating it by 90°.

```
\tikz \shadedraw [shading=axis,shading angle=90] (0,0) rectangle (1,1);
```

You can also define new shading types yourself. However, for this, you need to use the basic layer directly, which is, well, more basic and harder to use. Details on how to create a shading appropriate for filling paths are given in Section 23.3.

8.4.2 Choosing a Shading Color

The following options can be used to change the colors used for shadings. When one of these options is given, the `shade` option is automatically selected and also the "right" shading.

- `top color`=⟨*color*⟩ This option prescribes the color to be used at the top in an `axis` shading. When this option is given, several things happen:

  1. The `shade` option is selected.
  2. The `shading=axis` option is selected.
  3. The middle color of the axis shading is set to the average of the given top color ⟨*color*⟩ and of whatever color is currently selected for the bottom.
  4. The rotation angle of the shading is set to 0.

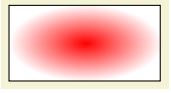
```
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

- `bottom color`=⟨*color*⟩ This option works like `top color`, only for the bottom color.
- `middle color`=⟨*color*⟩ This option specifies the color for the middle of an axis shading. It also sets the `shade` and `shading=axis` options, but it does not change the rotation angle.

  Note: Since both `top color` and `bottom color` change the middle color, this option should be given *last* if all of these options need to be given:

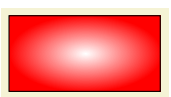
```
\tikz \draw[top color=white,bottom color=black,middle color=red]
  (0,0) rectangle (2,1);
```

- `left color`=⟨*color*⟩ This option does exactly the same as `top color`, except that the shading angle is set to 90°.
- `right color`=⟨*color*⟩ Works like `left color`.
- `inner color`=⟨*color*⟩ This option sets the color used at the center of a `radial` shading. When this option is used, the `shade` and `shading=radial` options are set.



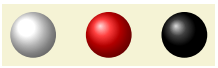
```
\tikz \draw[inner color=red] (0,0) rectangle (2,1);
```

- `outer color`=⟨*color*⟩ This option sets the color used at the border and outside of a `radial` shading.



```
\tikz \draw[outer color=red,inner color=white]
  (0,0) rectangle (2,1);
```

- `ball color=`⟨*color*⟩ This option sets the color used for the ball shading. It sets the `shade` and `shading=ball` options. Note that the ball will never "completely" have the color ⟨*color*⟩. At its "highlite" spot a certain amount of white is mixed in, at the border a certain amount of black. Because of this, it also makes sense to say `ball color=white` or `ball color=black`



```
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
\end{tikzpicture}
```

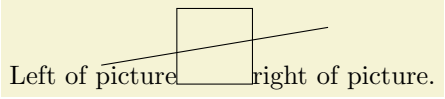
8.5 Establishing a Bounding Box

PGF is quite good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like "do not count this for the picture size" or "the picture is actually a little large." For this, you can use the option `use as bouding box` or the command `\useasboundingbox`, which is just a shorthand for `\path[use as bouding box]`.

- `use as bounding box` Normally, when this option is given on a path, the bounding box of the present path is used to determine the size of the picture and the size of all *subsequent* path commands are ignored. However, if there were previous path commands that have already established a larger bounding box, it will not be made smaller by this command.

  In a sense, `use as bounding box` has the same effect as clipping all subsequent drawing against the current path—without actually doing the clipping, only making PGF treat everything as if it were clipped.

  The first application of this command is to have a `{tikzpicture}` overlap with the main text:




```
Left of picture\begin{tikzpicture}
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,.25) -- (4,.75);
\end{tikzpicture}right of picture.
```

  In a second application, this command can be used to get better control over the white space around the picture:


```
Left of picture
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle (.5cm);
\end{tikzpicture}
right of picture.
```

  Note: If this option is used on a path inside a TEX group (scope), the effect "lasts" only till the end of the scope.

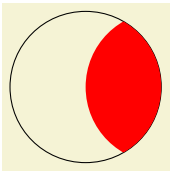
8.6 Using a Path For Clipping

To use a path for clipping, use the `clip` option.

- `clip` This option causes all subsequent drawings to be clipped against the current path and the size of subsequent paths will not be important for the picture size. If you clip against a self-intersecting path, the even-odd rule or the nonzero winding number rule is used to determine whether a point is inside or outside the clipping region.

  The clipping path is a normal graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all

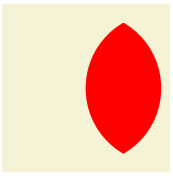
clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping area is to end a `{scope}`.



```
\begin{tikzpicture}
  \draw[clip] (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

It is usually a *very* good idea to apply the `clip` option only to the first path command in a scope.

If you "only wish to clip" and do not wish to draw anything, you can use the `clip` option together with the `\path` command or, which might be clearer, with the `\useasboundingbox` command. The effect is the same.



```
\begin{tikzpicture}
  \useasboundingbox[clip] (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

To keep clipping local, use `{scope}` environments as in the following example:

```
\begin{tikzpicture}
  \draw (0,0) -- ( 0:1cm);
  \draw (0,0) -- (10:1cm);
  \draw (0,0) -- (20:1cm);
  \draw (0,0) -- (30:1cm);
  \begin{scope}[fill=red]
    \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

    \draw (0,0) -- (40:1cm);
    \draw (0,0) -- (50:1cm);
    \draw (0,0) -- (60:1cm);
  \end{scope}
  \draw (0,0) -- (70:1cm);
  \draw (0,0) -- (80:1cm);
  \draw (0,0) -- (90:1cm);
\end{tikzpicture}
```

There is a slightly annoying catch: You cannot specify certain graphic options for the command used for clipping. For example, in the above code we could not have moved the `fill=red` to the `\fill` command. The reasons for this have to do with the internals of the PDF specification. You do not want to know the details, believe me. It is best simply not to specify any options for these commands.

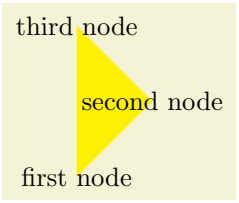
9 Nodes

9.1 Nodes and their Shapes

Ti*k*Z offers an easy way of adding so-called *nodes* to your pictures. In the simplest case, a node is just some text that is placed at some coordinate. However, a node can also have a border drawn around it or have a more complex background. Indeed, some nodes do not have a text at all, but solely consist of the background. You can name nodes so that you can reference their coordinates later in the picture. However, *nodes cannot be referenced across different pictures.*

There are no special TEX commands for adding a node to a picture; rather, there is path operation called `node` for this. Nodes are created whenever Ti*k*Z encounters `node` or `coordinate` at any point on a path where it would expect a normal path command (like `-- (1,1)` or `sin (1,1)`).

The node operation is typically followed by some options, which apply only to the node. Then, you can optionally *name* the node by providing a name in round braces. Lastly, for the `node` operation you must provide some label text for the node in curly braces, while for the `coordiante` operation you may not. The node is placed at the current position of the path *after the path has been drawn.* Thus, all nodes are drawn "on top" of the path and retained until the path is complete. If there are several nodes on a path, they are drawn on top of the path in the order they are encountered.




```
\tikz \fill[fill=yellow]
    (0,0) node {first node}
  -- (1,1) node {second node}
  -- (0,2) node {third node};
```

There are two possible syntax for specifying nodes:

- `node[`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`){`⟨*text*⟩`}` As with normal paths, you can give multiple options in multiple brackets. Also, you can change the ordering of ⟨*name*⟩ and ⟨*options*⟩. For example, the following is a legal node specification: `node[red](A)[draw]`.

  The effect of `at` is to place the node at the coordinate given after `at` and not, as would normally be the case, at the last position. The `at` syntax is not available when a node is given inside a path operation (it would not make any sense, there).

- `coordinate[`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`)` This has the same effect as

  `node[shape=coordinate][`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`){}`,

  where the `at` part might be missing. Note that you *must* give the options first.

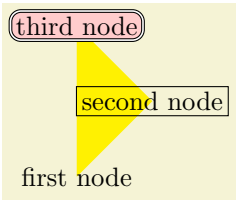
A node specification must be given on a path at some point where Ti*k*Z expects a path command. Node specification can also be given *inside* certain path commands; this is explained later.

The `(`⟨*name*⟩`)` is a name for later reference and it is optional. You also add the option `name=`⟨*name*⟩ to the ⟨*option*⟩ list; it has the same effect.

- `name=`⟨*node name*⟩ assigns a name to the node for later reference. Since this is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just `1` or perhaps `start of chart` or even `y_1`. Your node name should *not* contain an punctuation like a dot, a comma or a colon since these are used to detect what kind of coordinate you mean when you reference a node.

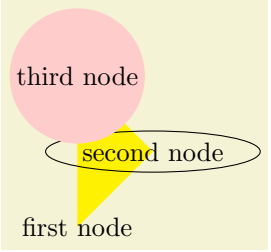
The ⟨*options*⟩ is an optional list of options that *apply only to the node* and have no effect outside. The other way round, most "outside" options also apply to the node, but not all. For example, the "outside" rotation does not apply to nodes (unless some special options are used). Also, the outside path action, like `draw` or `fill`, never applies to the node and must be given in the node (unless some special other options are used).

As mentioned before, we can add a border and even a background to a node:




```
\tikz \fill[fill=yellow]
      (0,0) node {first node}
   -- (1,1) node[draw] {second node}
   -- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

The "border" is actually just a special case of a much more general mechanism. Each node has a certain *shape* which, by default, is a rectangle. However, we can also ask Ti*k*Z to use circle shape instead or an ellipse shape (you have to inlude `pgflibraryshapes` for this shape):




```
\tikz \fill[fill=yellow]
      (0,0) node{first node}
   -- (1,1) node[ellipse,draw] {second node}
   -- (0,2) node[circle,fill=red!20] {third node};
```

In the future, there might be much more complicated shapes available such as, say, a shape for a resistor or a shape for a state of a finite automaton or a shape for a UML class. Unfortunately, creating new shapes is a bit tricky and makes it necessary to use the basic layer directly. Life is hard.

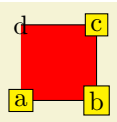
To select the shape of a node, the following option is used:

- `shape`=⟨*shape name*⟩ select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

  Since this option is used often, you can leave out the `shape=`. In detail, when Ti*k*Z encounters an option like `circle` that it does not know, it will, after everything else has failed, check whether this option is the name of some shape. If so, that shape is selected as if you had said `shape=`⟨*shape name*⟩.

  By default, the following shapes are available: `rectangle`, `circle`, `coordinate`, and, when the package `pgflibraryshapes` is loaded, also `circle`. Details of these shapes, like their anchors and size options, are discussed in Section 9.7.

- `shape action`=⟨*action list*⟩ presets which actions should be taken on the path that makes up the shape's background. By default, nothing is done with the path (it is thrown away) and you need to an option like `draw` inside the node's options to make Ti*k*Z actually draw the node. This can be bothersome if you have lots of nodes, all of which need to be drawn and filled. In this case, you can say `shape action={draw,fill}` in the enclosing scope to preset these two actions. A node can reset the actions by giving the `path only` option.

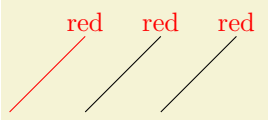



```
\begin{tikzpicture}[shape action={draw,fill=yellow},fill=red]
  \filldraw (0,0) node{a} --(1,0) node{b}
            --(1,1) node{c} --(0,1) node[path only]{d};
\end{tikzpicture}
```

9.2 Options For the Text in Nodes

The simplest option for the text in nodes is its color. Normally, this color is just the last color installed using `color=`, possibly inherited from another scope. However, it is possible to specificly set the color used for text using the following option"

- `text`=⟨*color*⟩ Sets the color to be used for text inside nodes. A `color=` option will immediately override this option.




```
\begin{tikzpicture}
  \draw[red]       (0,0) -- +(1,1) node[above]     {red};
  \draw[text=red]  (1,0) -- +(1,1) node[above]     {red};
  \draw            (2,0) -- +(1,1) node[above,red] {red};
\end{tikzpicture}
```

Normally, when a node is typset, all the text you give in the braces (without the options and the node name, of course) is but in one long line (in an `\hbox`, to be precise) and the node will become as wide as necessary.

You can change this behaviour using the following options. They allow you to limit the width of a node (naturally, at the expence of its height).

- `text width`=⟨*dimension*⟩ This option will put the text of a node in a box of the given width (more precisely, in a `{minipage}` of this width; for plain TEX a little "minipage emulation" is used).

  If the node text is not as wide as ⟨*dimension*⟩, it will nevertheless be put in a box of this width. If it is larger, line breaking will be done.

  By default, when this option is given, a ragged right border will be used. This is sensible since, typically, these boxes are narrow and justifying the text looks ugly.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm]
  {This is a demonstration text for showing how line breaking works.};
```

- `text justified` causes the text to be justified instead of (right)ragged. Use this only with pretty broad nodes.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm,text justified]
  {This is a demonstration text for showing how line breaking works.};
```

  In the above example, TEX complains (rightfully) about three very badly typeset lines.

- `text ragged` causes the text to be typeset with a ragged right. This uses the original plain TEX definition of a ragged right border, in which TEX will try to balance the right border as well as possible. This is the default.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm,text ragged]
  {This is a demonstration text for showing how line breaking works.};
```

- `text badly ragged` causes the right border to be ragged in the LATEX-style, in which no balancing occurs. This looks ugly, but it may be useful for very narrow boxes and when you wish to avoid hyphenations.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm,text badly ragged]
  {This is a demonstration text for showing how line breaking works.};
```

- `text centered` centers the text, but tries to balance the lines.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm,text centered]
  {This is a demonstration text for showing how line breaking works.};
```

- `text badly centered` centers the text, without balancing the lines.

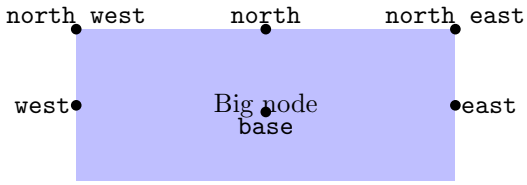
This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=yellow,text width=3cm,text badly centered]
  {This is a demonstration text for showing how line breaking works.};
```

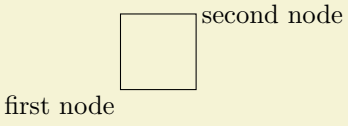
9.3 Placing Nodes Using Anchors

When you place a node at some coordinate, the node is centered on this coordinate by default. This is often undesirable and it would be better to have the node to the right or above the actual coordinate.

PGF uses a so-called anchoring mechanism to give you a very fine control over the placement. The idea is simple: Imaging a node of rectangular shape of a certain size. PGF defines numerous anchor positions in the shape. For example to upper right corner is called, well, not upper right anchor, but the `north east` anchor of the shape. The center of the shape has an anchor called `center` on top of it, and so on. Here are some examples (a complete list is given in Section 9.7).



Now, when you place a node at a certain coordinate, you can ask Ti*k*Z to place the node is shifted around in such a way that a certain anchor is at the coordinate. In the following example, we ask Ti*k*Z to shift the first node such that its `north east` anchor is at coordinate `(0,0)` and that the `west` anchor of the second node is at coordinate `(1,1)`.



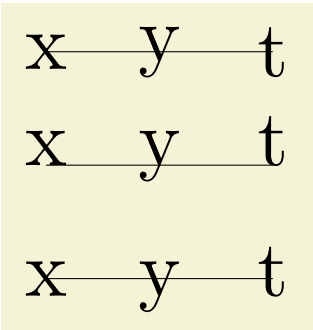

```
\tikz \draw           (0,0) node[anchor=north east] {first node}
          rectangle (1,1) node[anchor=west] {second node};
```

Since the default anchor is `center`, the default behaviour is to shift the node in such a way that it is centered on the current position.

- `anchor`=⟨*anchor name*⟩ causes the node to be shifted such that it's anchor ⟨*anchor name*⟩ lies on the current coordinate.

  The only anchor that is present in all shapes is `center`. However, most shapes will at least define anchors in all "compass directions." Furthermore, the standard shapes also define a `base` anchor, as well as `base west` and `base east`, for placing things on the baseline of the text.

  The standard shapes also define a `mid` anchor (and `mid west` and `mid east`). This anchor is half the height of the character "x" above the base line. This anchor is useful for vertically centering multiple nodes that have different heights and depth. Here is an example:




```
\begin{tikzpicture}[scale=3,transform shape]
  % First, center alignment -> whobbles
  \draw[anchor=center] (0,1)  node{x} -- (0.5,1)  node{y} -- (1,1)  node{t};
  % Second, base alignment -> no whobble, but too high
  \draw[anchor=base]   (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
  % Third, mid alignment
  \draw[anchor=mid]    (0,0)  node{x} -- (0.5,0)  node{y} -- (1,0)  node{t};
\end{tikzpicture}
```

Unfortunately, while perfectly logical, it is often rather counter-intuitive that in order to place a node *above* a given point, you need to specify the `south` anchor. For this reason, there are some useful options that allow you to select the standard anchors more intuitively:

- `above=`⟨*offset*⟩ does the same as `anchor=south`. If the ⟨*offset*⟩ is specified, the node is additionally shifted upwards by the given ⟨*offset*⟩.

above
```
\tikz \fill (0,0) circle (2pt) node[above] {above};
```

above
```
\tikz \fill (0,0) circle (2pt) node[above=2pt] {above};
```

- `above left=`⟨*offset*⟩ does the same as `anchor=south east`. If the ⟨*offset*⟩ is specified, the node is additionally shifted upwards and right by ⟨*offset*⟩.

above left
```
\tikz \fill (0,0) circle (2pt) node[above left] {above left};
```

above left
```
\tikz \fill (0,0) circle (2pt) node[above left=2pt] {above left};
```

- `above right=`⟨*offset*⟩ does the same as `anchor=south west`.

above right
```
\tikz \fill (0,0) circle (2pt) node[above right] {above right};
```

- `left=`⟨*offset*⟩ does the same as `anchor=east`.

left
```
\tikz \fill (0,0) circle (2pt) node[left] {left};
```

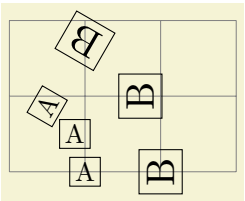
- `right=`⟨*offset*⟩ does the same as `anchor=west`.
- `below=`⟨*offset*⟩ does the same as `anchor=north`.
- `below left=`⟨*offset*⟩ does the same as `anchor=north east`.
- `below right=`⟨*offset*⟩ does the same as `anchor=north west`.

9.4 Transformations and Nodes

It is possible to transform nodes, but, by default, transformations do not apply to nodes. The reason is that you usually do *not* want your text to be scaled or rotated even if the main graphic is transformed. Scaling text is evil, rotating slightly less so.

However, sometimes you *do* wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways in which you can achieve this:

1. You can use the following option:
   - `transform shape` causes the current "external" transformation matrix to be applied to the shape. For example, if you said `\tikz[scale=3]` and then say `node[transform shape] {X}`, you will get a "huge" X in your graphic.
2. You can give transformation command *inside* the option list of the node. *These* transformations allways apply to the node.




```
\begin{tikzpicture}[shape action=draw]
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (1,0) node{A}
                          (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=30]        (1,0) node{A}
                          (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=60]        (1,0) node[transform shape] {A}
                          (2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

9.5 Placing Nodes on a Line or Curve

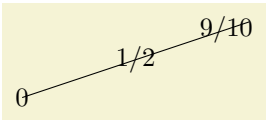
Until now, we always placed node on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on "the middle" of a line and we do not wish to compute these coordinates "by hand." To facilitate such placements, Ti*k*Z allows you to specify that a certain node should be somewhere "on" a line. There are two ways of specifying this: Either explicitly by using the `pos` option or implicitly by placing the node "inside" a path command. These two ways are described in the following.

9.5.1 Explicit Use of the Position Option

- `pos`=⟨*fraction*⟩ When this option is given, the node is not anchored on the last coordinate. Rather, it is anchored on some point on the line from the previous coordinate to the current point. The ⟨*fraction*⟩ dictates how "far" on the line the point should be. A ⟨*fraction*⟩ or 0 is the previous coordinate, 1 is the current one, everything else is in between. In particular, 0.5 is the middle.

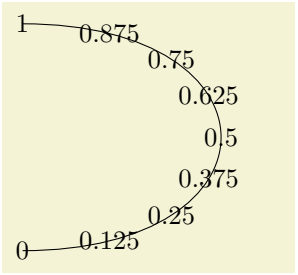
  Now, what is "the previous line"? This depends on the previous path construction command.

  In the simplest case, the previous path command was a "lineto" command, that is, a `--`⟨*coordinate*⟩ command:



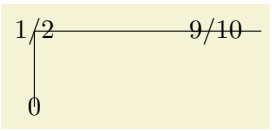

```
\tikz \draw (0,0) -- (3,1)
    node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

  The next case is the curveto command (the `..` command). In this case, the "middle" of the curve, that is, the position `0.5` is not necessarily the point at the exact half distance on the line. Rather, it is some point at "time" 0.5 of a point travelling from the start of the curve, where it is at time 0, to the end of the curve, which it reaches at time 0.5. The "speed" of the curve depends on the length of the support vectors (the vectors that connect the start and end points to the control points). The exact math is a bit complicated, you may wish to consult a good book on computer graphics and Beziér curves if you are intrigued.

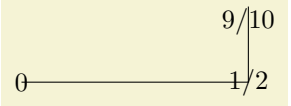



```
\tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3)
  \foreach \p in {0,0.125,...,1} {node[pos=\p]{\p}};
```

  Another interesting case are the horizontal/vertical lineto commands `|-` and `-|`. For them, the position (or time) `0.5` is exactly the corner point.



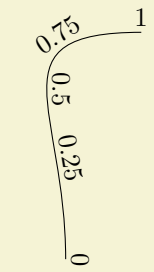

```
\tikz \draw (0,0) |- (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```



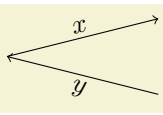

```
\tikz \draw (0,0) -| (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

  For all other path construction commands, *the position placement does not work*, currently. This will hopefulle change in the future (especially for the arc operation).

- `sloped` This option causes the node to be rotated such that a horizontal line becomes a tangent to the curve. The rotation will always be done in such a way that text is never "upside down." If you really need upside down text, use `[rotate=180]`.



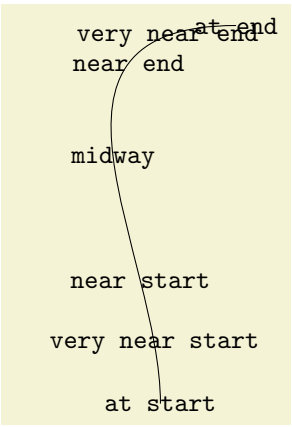

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    \foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```




```
\begin{tikzpicture}[->]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

There exist styles for specifying positions a bit less "technically":

- `style=midway` is set to `pos=0.5`.




```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:3cm) .. (1,5)
      node[at end]          {|at end|}
      node[very near end]   {|very near end|}
      node[near end]        {|near end|}
      node[midway]          {|midway|}
      node[near start]      {|near start|}
      node[very near start] {|very near start|}
      node[at start]        {|at start|};
```

- `style=near start` is set to `pos=0.25`.
- `style=near end` is set to `pos=0.75`.
- `style=very near start` is set to `pos=0.125`.
- `style=very near end` is set to `pos=0.875`.
- `style=at start` is set to `pos=0`.
- `style=at end` is set to `pos=1`.

9.5.2 Implict Use of the Position Option

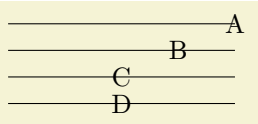
When you wish to place a node on the line `(0,0) -- (1,1)`, it is natural to specify the node not following the `(1,1)`, but "somewhere in the middle." This is, indeed, possible and you can write `(0,0) -- node{a} (1,1)` to place a node midway between `(0,0)` and `(1,1)`.

What happens is the following: The syntax of the lineto path command is actually `-- node`⟨*node specification*⟩⟨*coordinate*⟩. (It is even possible to give multiple nodes in this way.) When the optional `node` is encountered, that is, when the `--` is directly followed by `node`, then the specification(s) are read and "stored away." Then, after the ⟨*coordinate*⟩ has finally been reached, they are inserted again, but with the `pos` option set.

There are two things to note about this: When a node specification is "stored," its catcodes become fixed. This means that you cannot use overly complicated verbatim text in them. If you really need, say, a verbatim text, you will have to put it in a normal node following the coordinate and add the `pos` option.

Second, which `pos` is chosen for the node? The position is inherited from the surrounding scope. However, this holds only for nodes specified in this implicit way. Thus, if you add the option `[near end]` a scope,

this does not mean that *all* nodes given in this scope will be put on near the end of lines. Only the nodes for which an implicit `pos` is added will be placed near the end. In essence, this is what you want. Here are some examples that should make this clearer:




```
\begin{tikzpicture}[near end]
  \draw (0cm,4em) -- (3cm,4em) node{A};
  \draw (0cm,3em) --           node{B}          (3cm,3em);
  \draw (0cm,2em) --           node[midway] {C} (3cm,2em);
  \draw (0cm,1em) -- (3cm,1em) node[midway] {D} ;
\end{tikzpicture}
```

Like the lineto command, the curveto command `..` also allows you to specify nodes "inside" the command. After both the first `..` and also after the second `..` you can place node specifications. Like for the `--` command, these will be collected and then reinserted after the command with the `pos` option set.

9.6 Connecting Nodes

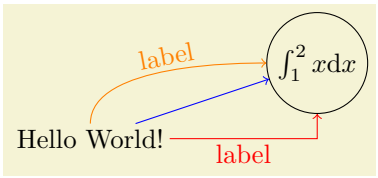
Once you have defined a node and given it a name, you can use this name to reference it. This can be done in two ways, see also Section 6.5. Suppose you have said `\path(0,0) node(x) {Hello World!};` in order to define a node named `x`.

1. Once the node `x` has been defined, you can use `(x.`⟨*anchor*⟩`)` whereever you would normally use a normal coordinate. This will yield the position at which the given ⟨*anchor*⟩ is in the picture. Note that transformations do not apply to this coordinate, that is, `(x.north)` will be the northern anchor of `x` even if you have said `scale=3` or `xshift=4cm`. This is usually what you would expect.

2. You can also just use `(x)` as a coordinate. In most cases, this gives the same coordinate as `(x.center)`. Indeed, if the `shape` of `x` is `coordinate`, then `(x)` and `(x.center)` have exactly the same effect.

   However, for most other shapes, some path construction commands like `--` try to be "clever" when this they are asked to draw a line from such a coordinate or to such a coordinate. When you say `(x)--(1,1)`, the `--` path command will not draw a line from the center of `x`, but *from the border* of `x` in the direction going towards `(1,1)`. Likewise, `(1,1)--(x)` will also have the line end on the border in the direction coming from `(1,1)`.

   In addition to `--`, the curveto path command `..` and the path commands `-|` and `|-` will also handle nodes without anchors correctly. Here is an example, see also Section 6.5:




```
\begin{tikzpicture}
  \path (0,0) node             (x) {Hello World!}
        (3,1) node[circle,draw](y) {$\int_1^2 x \mathrm d x$};

  \draw[->,blue]   (x) -- (y);
  \draw[->,red]    (x) -| node[near start,below] {label} (y);
  \draw[->,orange] (x) .. controls +(up:1cm) and +(left:1cm) .. node[above,sloped] {label} (y);
\end{tikzpicture}
```

9.7 The Predefined Shapes

PGF and Ti*k*Z define three shapes, by default:

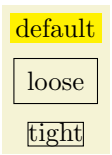
- `rectangle`,
- `circle`, and
- `coordinate`.

By loading library packages, you can define more shapes. Currently, the package `pgflibraryshapes` defines

- `ellipse`.

The exact behaviour of these shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviours. However, there are some options that apply to most shapes:

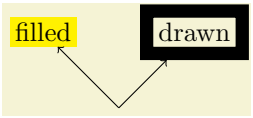
- `inner sep`=⟨*dimension*⟩ An additional (invisible) separation space of ⟨*dimension*⟩ will be added inside the shape, between the text and the shape's background path. The effect is as if you had added appropriate horizontal and vertical skips at the beginning and end of the text to make it a bit "larger."




```
\begin{tikzpicture}
  \draw (0,0)     node[inner sep=0pt,draw]        {tight}
        (0cm,2em) node[inner sep=5pt,draw]        {loose}
        (0cm,4em) node[inner sep=2pt,fill=yellow] {default};
\end{tikzpicture}
```

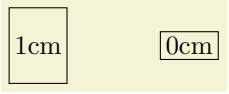
- `inner xsep`=⟨*dimension*⟩ Specifies the inner separation in the x-direction, only.
- `inner ysep`=⟨*dimension*⟩ Specifies the inner separation int the y-direction, only.
- `outer sep`=⟨*dimension*⟩ This option adds an additional (invisible) separation space of ⟨*dimension*⟩ outside the background path. The main effect of this option is that all anchors will move a little "to the outside."

  The default for this option is half the line width. When the default is used and when the background path is draw, the anchors will lie exactly on the "outside border" of the path (not on the path itself). When the shape is filled, but not drawn, this may not be desirable. In this case, the `outer sep` should be set to zero point.



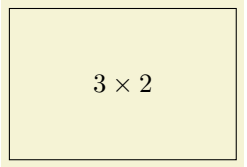

```
\begin{tikzpicture}
  \draw[line width=5pt]
    (0,0) node[outer sep=0pt,fill=yellow]          (f) {filled}
    (2,0) node[inner sep=.5\pgflinewidth+2pt,draw] (d) {drawn};

  \draw[->] (1,-1) -- (f);
  \draw[->] (1,-1) -- (d);
\end{tikzpicture}
```

- `outer xsep`=⟨*dimension*⟩ Specifies the outer separation in the x-direction, only.
- `outer ysep`=⟨*dimension*⟩ Specifies the outer separation int the y-direction, only.
- `minimum height`=⟨*dimension*⟩ This option ensures that the height of the shape (including the inner, but ignoring the outer separation) will be at least ⟨*dimension*⟩. Thus, if the text plus the inner separation is not at least as large as ⟨*dimension*⟩, the shape will be enlarged appropriately. However, if the text is already larger than ⟨*dimension*⟩, the shape will not be shrunk.



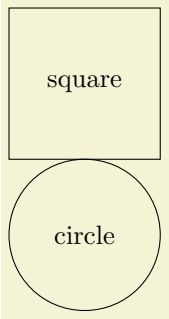

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=1cm,draw] {1cm}
        (2,0) node[minimum height=0cm,draw] {0cm};
\end{tikzpicture}
```

- `minimum width`=⟨*dimension*⟩ same as `minimum height`, only for the width.



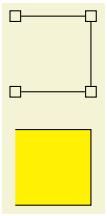

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

- `minimum size`=⟨*dimension*⟩ sets both the minimum height and width at the same time.




```
\begin{tikzpicture}
  \draw (0,0)  node[minimum size=2cm,draw] {square};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {circle};
\end{tikzpicture}
```

The `coordinate` shape is handled in a special way by Ti*k*Z. When a node `x` whose shape is `coordinate` is used as a coordinate `(x)`, this has the same effect as if you had said `(x.center)`. None of the special "line shortening rules" apply in this case. This can be useful since, normally, the line shortening causes paths to be segmented and they cannot be used for filling. Here is an example that demonstrates the difference:



```
\begin{tikzpicture}
  \path[yshift=1.5cm,shape=rectangle,shape action=draw]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=yellow] (a1) -- (a2) -- (a3) -- (a4);

  \path[shape=coordinate,shape action=draw]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) coordinate(b3) (0,1) coordinate(b4);
  \filldraw[fill=yellow] (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

10 Transformations

PGF has a powerful transformation mechanism that is similar to the transformation capabilities of METAFONT. The present section explains how you can access it in Ti*k*Z.

10.1 The Different Coordinate Systems

It is a long process from a coordinate like, say, $(1, 2)$ or $(1cm, 5pt)$, until a point is finally placed on the display. In order to find out where the point should go, it is constantly "transformed," which means that it is mostly shifted around and possibly rotated, slanted, scaled, and otherwise mutilated.

In detail, (at least) the following transformations are applied to a coordiante like $(1, 2)$ before a point on the screen is chosen:

1. PGF interprets a coordinate like $(1, 2)$ in its xy-coordinate sytem as "add the current x-vector once and the current y-vectore twice to obtain the new point."
2. PGF applies its coordinate transformation matrix to the resulting coordinate. This yields the final position of the point inside the picture.
3. The backend driver (like `dvips` or `pdftex`) adds transformation commands such the coordinate is shifted to the correct position on the page.
4. PDF (or PostScript) apply the canvas transformation matrix to the point, which can once more change the position on the page.
5. The viewer application or the printer applies the device transformation matrix to transform the coordinate to its final pixel coordinate on the screen or paper.

In reality, the process is even more involved, but the above should give the idea: A point is constantly transformed by changes of the coordinate system.

In Ti*k*Z, you only have access to the first two coordiante systems: The xy-coordinate system and the coordinate transformation matrix (these will be explained later). PGF also allows you to change the canvas transformation matrix, but you have to use commands of the core layer directly to do so and you "better know what you are doing" when you use do this. The moment you start modifying the canvas matrix, PGF immediately looses track of all coordinates and shapes, anchors, and bounding box computations will no longer work.

10.2 The xy- and xyz-Coordinate Systems

The first and easiest coordinate systems are PGF's xy- and xyz-coordinate systems. The idea is very simple: Whenever you specify a coordinate like `(2,3)` this means $2v_x + 3v_y$, where v_x is the current *x-vector* and v_y is the current *y-vector*. Similarly, the coordinate `(1,2,3)` means $v_x + 2v_y + 3v_z$.

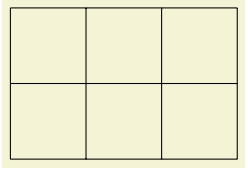
Unlike other packages, PGF does not insist that v_x actually has a y-component of 0, that is, that it is a horizontal vector. Instead, the x-vector can point anywhere you want. Naturally, *normally* you will want the x-vector to point horizontallly.

One undesirable effect of the flexibility offered by the fact that the x-vector does not need to point in the x-direction is that it is not possible to provide mixed coordinates as in $(1, 2pt)$. Life is hard.

To change the x-, y-, and z-vectors, you can use the following options:

- `x`=⟨*dimension*⟩ Sets the x-vector of PGF's xyz-coordinate system to point ⟨*dimension*⟩ to the right, that is, to $(\langle dimension\rangle, 0pt)$. The default is 1cm.

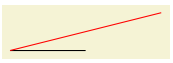

```
\begin{tikzpicture}
  \draw                  (0,0)   -- +(1,0);
  \draw[x=2cm,color=red] (0,0.1) -- +(1,0);
\end{tikzpicture}
```



```
\tikz \draw[x=1.5cm] (0,0) grid (2,2);
```

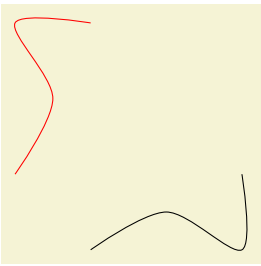
The last example shows that the size of steppings in grids, just like all other dimensions, are not affected by the x-vector. After all, the x-vector is only used to determine the coordinate of the upper right corner of the grid.

- x=⟨*coordinate*⟩ Sets the x-vector of PGF's xyz-coordinate system to the specified ⟨*coordinate*⟩. If ⟨*coordinate*⟩ contains a comma, it must be put in braces.



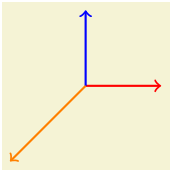
```
\begin{tikzpicture}
  \draw                              (0,0) -- (1,0);
  \draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

You can use this, for example, to exchange the meaning of the x- and y-coordinate.



```
\begin{tikzpicture}[smooth]
  \draw plot ((1,0) (2,0.5) (3,0) (3,1));
  \draw[x={(0cm,1cm)},y={(1cm,0cm)},color=red]
        plot ((1,0) (2,0.5) (3,0) (3,1));
\end{tikzpicture}
```

- y=⟨*value*⟩ Works like the `x=` option, only if ⟨*value*⟩ is a dimension, the resulting vector points to $(0, \langle value \rangle)$.
- z=⟨*value*⟩ Works like the `z=` option, but now a dimension is means the point $(\langle value \rangle, \langle value \rangle)$.

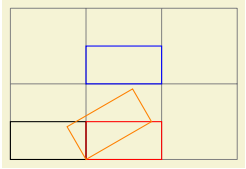


```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0);
  \draw[color=blue] (0,0,0) -- (0,1,0);
  \draw[color=orange] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

10.3 Coordinate Transformation

PGF and Ti*k*Z allow you to specify *coordiante transformations*. Whenever you specify a coordinate as in `(1,0)` or `(1cm,1pt)` or `(A.north)` or `(30:2cm)`, this coordinate is first "reduced" to a position of the form "x points to the right and y points upwards." For example, `(1in,5pt)` is reduced to "$72\frac{72}{100}$ points to the right and 5 points upwards" and `(90:100pt)` means "0pt to the right and 100 points upwards."

The next step is to apply the current *coordinate transformation matrix* to the coordinate. For example, the coordinate transformation matrix might currently be set such that it adds a certain constant to the x value. Also, it might be setup such that it, say, exchanges the x and y value. In general, any "standard" transformtion like translation, rotation, slanting, or scaling or any combination thereof is possible. (Internally, PGF keeps track of a coordinate transformation matrix very much like the concatentation matrix used by PDF or PostScript.)



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw (0,0) rectangle (1,0.5);
  \begin{scope}[xshift=1cm]
    \draw             [red]    (0,0) rectangle (1,0.5);
    \draw[yshift=1cm] [blue]   (0,0) rectangle (1,0.5);
    \draw[rotate=30]  [orange] (0,0) rectangle (1,0.5);
  \end{scope}
\end{tikzpicture}
```

The most important aspect of the coordinate transforamtion matrix is *that it applies to coordiantes only!* In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle. In certain cases, it is not immediately clear whether the coordinate transformation matrix *should* apply to a certain dimension. For example, should the coordinate transformation matrix apply to grids? (It does.) And what about the size of arced corners? (It does not.) The general rule is "If there is

no ‘coordinate’ involved, even ‘indirectly,’ the matrix is not applied.” However, sometimes, you simply have to try or look it up in the documentation whether the matrix will be applied.

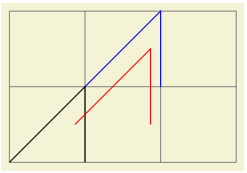
Setting the matrix cannot be done directly. Rather, all you can do is to “add” another transformation to the current matrix. However, all transformations are local to the current TEX-group. All transformations are added using graphic options, which are described below.

Note that transformations apply immediately when they are encountered “in the middle of a path” and they apply only to the coordinates on the path following the transformation option.

```
\tikz \draw (0,0) rectangle (1,0.5) [xshift=2cm] (0,0) rectangle (1,0.5);
```

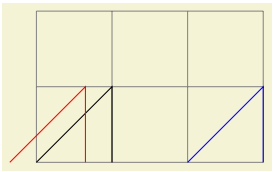
A final word of warning: You should refrain from using “aggressive” transformations like a scaling of a factor of 10000. The reason is that all transformations are done using TEX, which has a fairly low accuracy. Furthermore, in certain situations it is necessary the Ti*k*Z *inverts* the current transformation matrix and this will fail if the transformation matrix is badly conditioned or even singular (if you do not know what singular matrices are, never mind).

- `shift={`⟨*coordinate*⟩`}` adds the ⟨*coordinate*⟩ to all coordiantes.



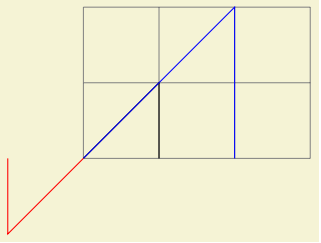
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                       (0,0) -- (1,1) -- (1,0);
  \draw[shift={(1,1)},blue]   (0,0) -- (1,1) -- (1,0);
  \draw[shift={(30:1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `xshift=`⟨*dimension*⟩ adds ⟨*dimension*⟩ to the x value of all coordiantes.



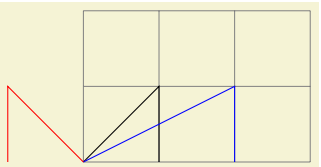
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (0,0) -- (1,1) -- (1,0);
  \draw[xshift=2cm,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xshift=-10pt,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `yshift=`⟨*dimension*⟩ adds ⟨*dimension*⟩ to the y value of all coordiantes.
- `scale=`⟨*factor*⟩ multiplies all coordinates by the given ⟨*factor*⟩. The ⟨*factor*⟩ should not be excessively large in absolute terms or very near to zero.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                (0,0) -- (1,1) -- (1,0);
  \draw[scale=2,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[scale=-1,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

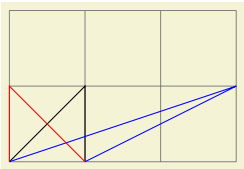
- `xscale=`⟨*factor*⟩ multiplies only the x-value of all coordinates by the given ⟨*factor*⟩.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                 (0,0) -- (1,1) -- (1,0);
  \draw[xscale=2,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xscale=-1,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

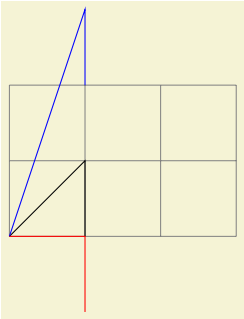
- `yscale=`⟨*factor*⟩ multiplies only the y-value of all coordinates by ⟨*factor*⟩.

- `xslant`=⟨*factor*⟩ slants the coordinate horizontally by the given ⟨*factor*⟩:



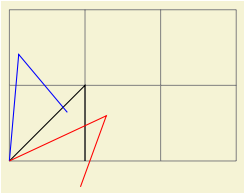
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                (0,0) -- (1,1) -- (1,0);
  \draw[xslant=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[xslant=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `yslant`=⟨*factor*⟩ slants the coordinate vertically by the given ⟨*factor*⟩:



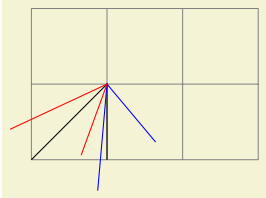
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                (0,0) -- (1,1) -- (1,0);
  \draw[yslant=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[yslant=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `rotate`=⟨*degree*⟩ rotates the coordinate system by ⟨*degree*⟩:



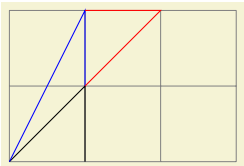
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                 (0,0) -- (1,1) -- (1,0);
  \draw[rotate=40,blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate=-20,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `rotate around`=`{`⟨*degree*⟩`:`⟨*coordinate*⟩`}` rotates the coordinate system by ⟨*degree*⟩ around the point ⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                                 (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={40:(1,1)},blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={-20:(1,1)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `cm`=`{`⟨*a*⟩`,`⟨*b*⟩`,`⟨*c*⟩`,`⟨*d*⟩`,`⟨*coordinate*⟩`}` applies the following transformation to all coordinates: Let (x, y) be the coordinate to be transformed and let ⟨*coordinate*⟩ specify the point (t_x, t_y). Then the new coordinate is given by $\left(\begin{smallmatrix} a & b \\ c & c \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} t_x \\ t_y \end{smallmatrix}\right)$. Usually, you do not use this option directly.



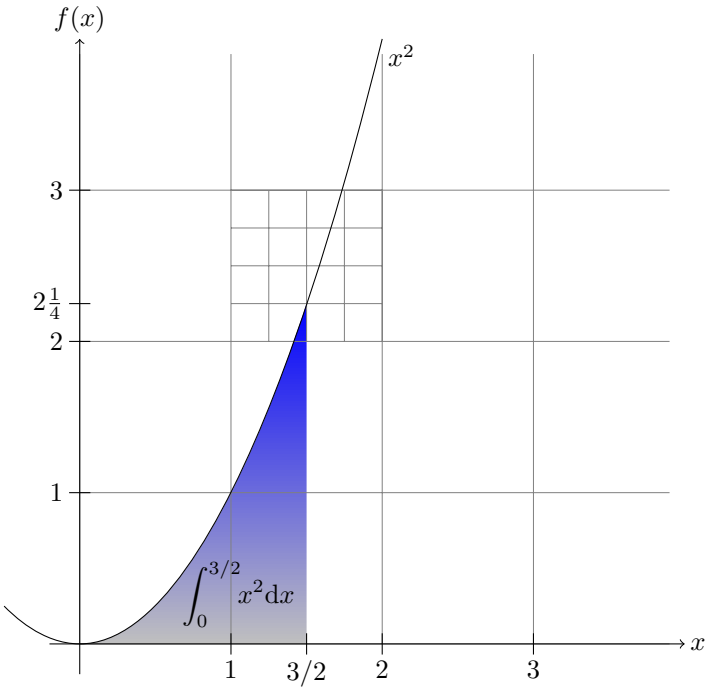
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                                 (0,0) -- (1,1) -- (1,0);
  \draw[cm={1,1,0,1,(0,0)},blue]    (0,0) -- (1,1) -- (1,0);
  \draw[cm={0,1,1,0,(1cm,1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `reset cm` completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes. Do not use this option.

Part III
Library and Utilities

In this part the library and utility packages are documented. The library packages provide additional predefined graphic objects like new arrow heads, or new plot marks. These are not loaded by default since many users will not need them.

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 right (1.5,2.25) -- (1.5,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \!\!x^2\mathrm{d}x$};

  \draw[style=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/{3/2}, 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 left (0,0) parabola right (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

11 Libraries

11.1 Arrow Tip Library

```
\usepackage{pgflibraryarrows} % LaTeX
\input pgflibraryarrows.tex   % plain TeX
\input pgflibraryarrows.tex   % ConTeX
```

The package defines additional arrow tips, which are described below. See page 138 for the arrows tips that are defined by default. Note that neither the standard packages nor this package defines an arrow name containing > or <. These are left for the user to defined as he or she sees fit.

11.1.1 Arrow Tips with Differing Names for the Left and Right Ends

`[-]` yields thick  and thin 

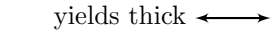
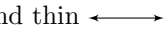
`]-[` yields thick  and thin 

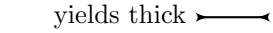
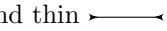
`(-)` yields thick  and thin 

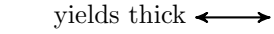
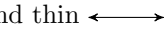
`)-(` yields thick  and thin 

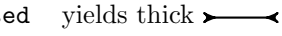
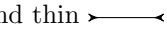
11.1.2 Variants of Other Arrow Tips

The same name is used for both the start and end arrows. Thus, for example, to install the first of the following arrow tips for both the start and the end, you would say `\pgfsetarrows{latex'-latex'}`:

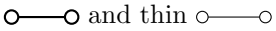
`latex'` yields thick  and thin 

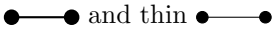
`latex' reversed` yields thick  and thin 

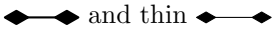
`stealth'` yields thick  and thin 

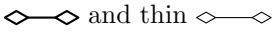
`stealth' reversed` yields thick  and thin 

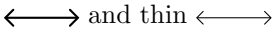
11.1.3 General Purpose Arrow Tips

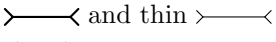
`o` yields thick  and thin

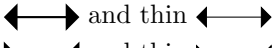
`*` yields thick  and thin

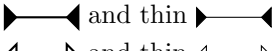
`diamond` yields thick  and thin

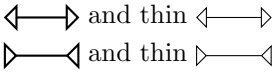
`open diamond` yields thick  and thin

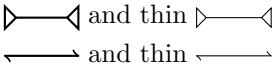
`angle 90` yields thick  and thin

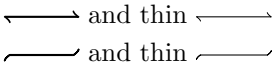
`angle 90 reversed` yields thick  and thin

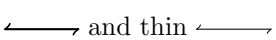
`triangle 90` yields thick  and thin

`triangle 90 reversed` yields thick  and thin

`open triangle 90` yields thick  and thin

`open triangle 90 reversed` yields thick  and thin

`left to` yields thick  and thin

`left to reversed` yields thick  and thin

`right to` yields thick and thin

`right to reversed` yields thick and thin

`hooks` yields thick and thin

`hooks reversed` yields thick and thin

`left hook` yields thick and thin

`left hook reversed` yields thick and thin

`right hook` yields thick and thin

`right hook reversed` yields thick and thin

11.1.4 Line Caps

`round cap`	yields for line width 1ex
`butt cap`	yields for line width 1ex
`triangle 90 cap`	yields for line width 1ex
`triangle 90 cap reversed`	yields for line width 1ex
`fast cap`	yields for line width 1ex
`fast cap reversed`	yields for line width 1ex

11.2 Plot Handler Library

```
\usepackage{pgflibraryplothandlers} % LaTeX
\input pgflibraryplothandlers.tex   % plain TeX
\input pgflibraryplothandlers.tex   % ConTeX
```

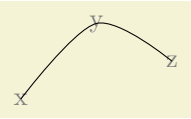
This library packages defines additional plot handlers, see Section 25.3 for an introduction to plot handlers. The additional handlers are described in the following.

11.2.1 Curve Plot Handlers

`\pgfplothandlercurveto`

This handler will issue a `\pgfpathcurveto` command for each point of the plot, *except* possibly for the first. As for the line-to handler, what happens with the first point can be specified using `\pgfsetmovetofirstplotpoint` or `\pgfsetlinetofirstplotpoint`.

Obviously, the `\pgfpathcurveto` command needs, in addition to the points on the path, some control points. These are generated automatically using a somewhat "dumb" algorithm: Suppose you have three points x, y, and z on the curve such that y is between x and z:

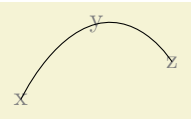



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

In order to determine the control points of the curve at the point y, the handler computes the vector $z - x$ and scales it by the tension factor (see below). Let us call the resulting vector s. Then $y + s$ and $y - s$ will be the control points around y. The first control point at the beginning of the curve will be the beginning itself, once more; likewise the last control point is the end itself.

`\pgfsetplottension{`⟨*value*⟩`}`

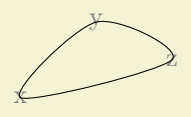
Sets the factor used by the curve plot handlers to determine the distance of the control points from the points they control. The default is 0.15. The higher the curvature of the curve points, the higher this value should be.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplottension{0.3}
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerclosedcurve

This handler works like the curve-to plot handler, only it will add a new part to the current path that is a closed curve through the plot points.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerclosedcurve
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

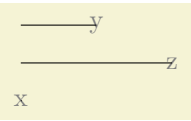
11.2.2 Comb Plot Handlers

There are three "comb" plot handlers. There name stems from the fact that the plots they produce look like "combs" (more or less).

\pgfplothandlerxcomb

This handler converts each point in the plot stream into a line from the y-axis to the point's coordinate, resulting in a "vertical comb."

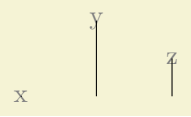
This handler is useful for creating "bar diagrams."




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerxcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerycomb

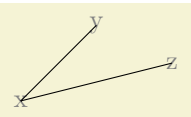
This handler converts each point in the plot stream into a line from the x-axis to the point's coordinate, resulting in a "vertical comb."




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerycomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerpolarcomb

This handler converts each point in the plot stream into a line from the origin to the point's coordinate.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerpolarcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

11.2.3 Mark Plot Handler

\pgfplothandlermark{⟨*mark code*⟩}

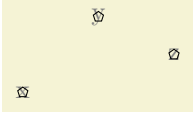
This command will execute the ⟨*mark code*⟩ for each point of the plot, but each time the coordinate transformation matrix will be setup such that the origin is at the position of the point to be plotted. This way, if the ⟨*mark code*⟩ draws a little circle around the origin, little circles will be drawn at each point of the plot.


```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

Typically, the ⟨*code*⟩ will be `\pgfuseplotmark{`⟨*plot mark name*⟩`}`, where ⟨*plot mark name*⟩ is the name of a predefined plot mark.

\pgfuseplotmark{⟨*plot mark name*⟩}

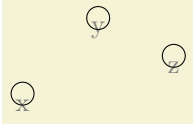
Draws the given ⟨*plot mark name*⟩ at the origin. The ⟨*plot mark name*⟩ must have been previously declared using `\pgfdeclareplotmark`.



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{pentagon}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfdeclareplotmark{⟨*plot mark name*⟩}{⟨*code*⟩}

Declares a plot mark for later used with the `\pgfuseplotmark` command.

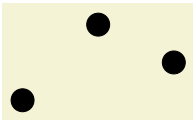


```
\pgfdeclareplotmark{my plot mark}
  {\pgfpathcircle{\pgfpoint{0cm}{1ex}}{1ex}\pgfusepathqstroke}
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{my plot mark}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetplotmarksize{⟨*dimension*⟩}

This command sets the TEX dimension `\pgfplotmarksize` to ⟨*dimension*⟩. This dimension is a "recommendation" for plot mark code at which size the plot mark should be drawn; plot mark code may choose to ignore this ⟨*dimension*⟩ altogether. For circles, ⟨*dimension*⟩ should be the radius, for other shapes it should be about half the width/height.

The predefined plot marks all take this dimension into account.



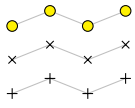
```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplotmarksize{1ex}
  \pgfplothandlermark{\pgfuseplotmark{*}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotmarksize

A TEX dimension that is a "recommendation" for the size of plot marks.

The following plot marks are predefined (the filling color has been set to yellow):

\pgfuseplotmark{*}
\pgfuseplotmark{x}
\pgfuseplotmark{+}

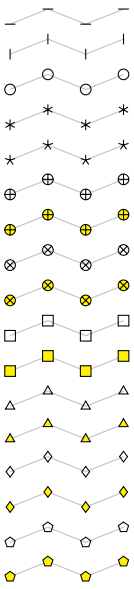


11.3 Plot Mark Library

```
\usepackage{pgflibraryplotmarks} % LaTeX
\input pgflibraryplotmarks.tex   % plain TeX
\input pgflibraryplotmarks.tex   % ConTeX
```

When this package is loaded, the following plot marks are defined in addition to `*`, `x`, and `+` (the filling color has been set to yellow):

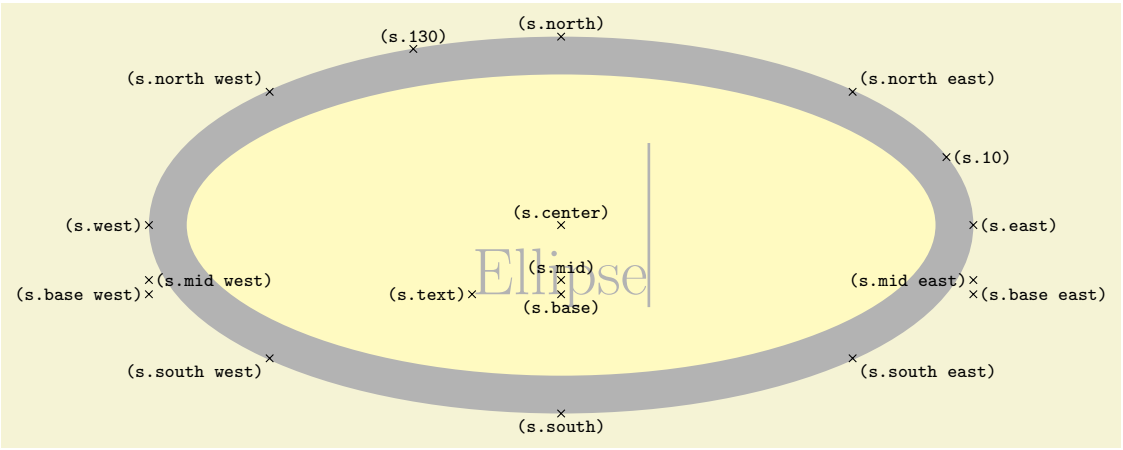
\pgfuseplotmark{-}
\pgfuseplotmark{|}
\pgfuseplotmark{o}
\pgfuseplotmark{asterisk}
\pgfuseplotmark{star}
\pgfuseplotmark{oplus}
\pgfuseplotmark{oplus*}
\pgfuseplotmark{otimes}
\pgfuseplotmark{otimes*}
\pgfuseplotmark{square}
\pgfuseplotmark{square*}
\pgfuseplotmark{triangle}
\pgfuseplotmark{triangle*}
\pgfuseplotmark{diamond}
\pgfuseplotmark{diamond*}
\pgfuseplotmark{pentagon}
\pgfuseplotmark{pentagon*}



11.4 Shape Library

Shape ellipse

This shape is an ellipse tightly fitting the text box, if not inner separation is given. The following figure shows the anchors this shape defines; the anchors `10` and `130` are example of border anchors.




```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse,style=shape example] {Ellipse\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] ((0,0)) node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

12 Repeating Things: The Foreach Statement

In this section the package `pgffor.sty` is described. It can be used independently of PGF, but it works particularly well together with PGF and Ti*k*Z.

When you say `\usepackage{pgffor}`, two commands are defined: `\foreach` and `\breakforeach`. Their behaviour is described in the following:

`\foreach` ⟨*variables*⟩ `in {`⟨*list*⟩`}` ⟨*commands*⟩

The syntax of this command is a bit complicated, so let us go through it step-by-step.

In the easiest case, ⟨*variables*⟩ is a single TEX-command like `\x` or `\point`. (If you want to have some fun, you can also use active characters. If you do not know what active characters are, you are blessed.)

Still in the easiest case, ⟨*list*⟩ is a comma-separated list of values. Anything can be used as a value, but numbers are most likely.

Finally, in the easiest case, ⟨*commands*⟩ is some TEX-text in curly braces.

With all these assumptions, the `\foreach` statement will execute the ⟨*commands*⟩ repeatedly, once for every element of the ⟨*list*⟩. Each time the ⟨*commands*⟩ are executed, the ⟨*variable*⟩ will be set to the current value of the list item.

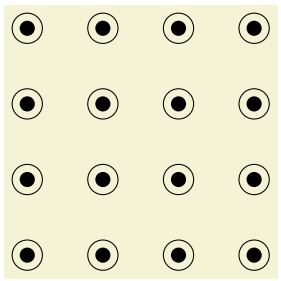
[1][2][3][0]

```
\foreach \x in {1,2,3,0} {[\x]}
```

Syntax for the commands. Let use move on to a more complicated setting. The first complication occurs when the ⟨*commands*⟩ are not some text in curly braces. If the `\foreach` statement does not encounter an opening brace, it will instead scan everything up to the next semicolon and use this as ⟨*commands*⟩. This is most useful in situations like the following:

```
\tikz
  \foreach \x in {0,1,2,3}
    \draw (\x,0) circle (0.2cm);
```

However, the "reading till the next semicolon" is not the whole truth. There is another rule: If a `\foreach` statement is directly followed by another `\foreach` statement, this second foreach statement is collected as ⟨*commands*⟩. This allows you to write the following:



```
\begin{tikzpicture}
  \foreach \x in {0,1,2,3}
    \foreach \y in {0,1,2,3}
      {
        \draw (\x,\y) circle (0.2cm);
        \fill (\x,\y) circle (0.1cm);
      }
\end{tikzpicture}
```

In general, the ⟨*commands*⟩ should be what you expect it to be.

The dots notation. The second complication concerns the ⟨*list*⟩. If this ⟨*list*⟩ contains the list item `...`, this list item is replaced by the "missing values." More precisely, the following happens:

Normally, when a list item `...` is encountered, there should already have been *two* list items before it, which where numbers. Examples of *numbers* are `1`, `-10`, or `-0.24`. Let us call these numbers x and y and let $d := y - x$ be their difference. Next, there should also be one number following the three dots, let us call this number z.

In this situation, the part of the list reading "x`,`y`,...,`z" is replaced by "x, $x + d$, $x + 2d$, $x + 3d$, ..., $x + md$," where the last dots are semantic dots, not syntactic dots. The value m is the largest number such that $x + md \leq z$ if d is positive or such that $x + md \geq z$ if d is negative.

Perhaps it is best to explain this by some examples: The following ⟨*list*⟩ have the same effects:

`\foreach \x in {1,2,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,2,3,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,3,...,11} {\x, }` yields 1, 3, 5, 7, 9, 11,

`\foreach \x in {1,3,...,10} {\x, }` yields 1, 3, 5, 7, 9,

`\foreach \x in {0,0.1,...,0.5} {\x, }` yields 0, 0.1, 0.20001, 0.30002, 0.40002,

`\foreach \x in {a,b,9,8,...,1,2,2.125,...,2.5} {\x, }` yields a, b, 9, 8, 7, 6, 5, 4, 3, 2, 1, 2, 2.125, 2.25, 2.375, 2.5,

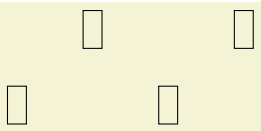
As can be seen, for fractional steps that are not multiples of 2^{-n} for some small n, rounding errors can occur pretty easily. Thus, in the second last case, `0.5` should probably be replaced by `0.501` for robustness.

There is yet another special case for the `...` statement: If the `...` is used right after the first item in the list, that is, if there is an x, but no y, the difference d obviously cannot be computed and is set to 1 if the number z following the dots is larger than x and is set to -1 if z is smaller:

`\foreach \x in {1,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

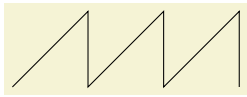
`\foreach \x in {9,...,3.5} {\x, }` yields 9, 8, 7, 6, 5, 4,

Special handling of pairs. Different list items are separated by commas. However, this causes a problem when the list items contain commas themselves as pairs like `(0,1)` do. In this case, you should put the items containing commas in braces as in `{(0,1)}`. However, since pairs are such a natural and useful case, they get a special treatment by the `\foreach` statement. When a list item starts with a `(` everything up to the next `)` is made part of the item. Thus, we can write things like the following:

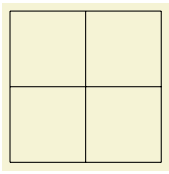


```
\tikz
  \foreach \position in {(0,0), (1,1), (2,0), (3,1)}
    \draw \position rectangle +(.25,.5);
```

Using the foreach-statement inside paths. Ti*k*Z allows you to use a `\foreach` statement inside a path construction. In such a case, the ⟨*commands*⟩ must be path construction commands. Here are two examples:



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



```
\tikz \draw \foreach \p in {1,...,3} {(\p,1)--(\p,3) (1,\p)--(3,\p)};
```

Multiple variables. You will often wish to iterate over two variables at the same time. Since you can nest `\foreach` loops, this is normally straight-forward. However, you sometimes wish variables to iterate "simultaneously." For example, we might be given a list of edges that connect two coordinates and might wish to iterate over these edges. While doing so, we would like the source and target of the edges to be set to two different variables.

To achieve this, you can use the following syntax: The ⟨*variables*⟩ may not only be a single TEX-variable. Instead, it can also be a list of variables separated by slahes (`/`). In this case the list items can also be lists of values separated by slashes.

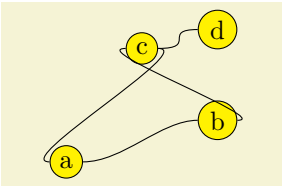
Assuming that the ⟨*variables*⟩ and the list items are lists of values, each time the ⟨*commands*⟩ are executed, each of the variables in ⟨*variables*⟩ is set to one part of the list making up the current list item. Here is an example to clarify this:

Example: `\foreach \x / \y in {1/2,a/b} {``\x\ and \y''}` yields "1 and 2" "a and b".

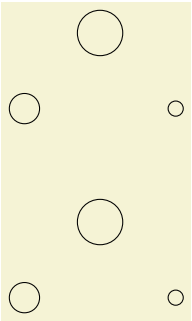
If some entry in the ⟨*list*⟩ does not have "enough" slashes, the last entry will be repeated. Here is an example:


```
\begin{tikzpicture}
  \foreach \x/\xtext in {0,...,3,2.72 / e}
    \draw (\x,0) node{$\xtext$};
\end{tikzpicture}
```

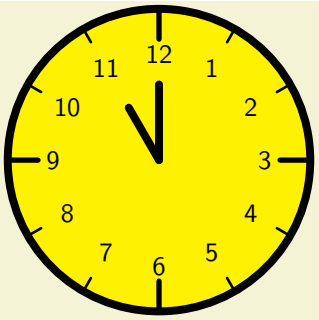
Here are more useful examples:



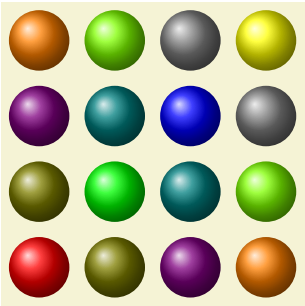

```
\begin{tikzpicture}
  % Define some coordinates:
  \path[shape=circle,shape action={draw,fill},fill=yellow]
    (0,0)    node(a) {a}
    (2,0.55) node(b) {b}
    (1,1.5)  node(c) {c}
    (2,1.75) node(d) {d};

  % Draw some connections:
  \foreach \source/\target in {a/b, b/c, c/a, c/d}
    \draw (\source) .. controls +(.75cm,0pt) and +(-.75cm,0pt)..(\target);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  % Let's draw circles at interesting points:
  \foreach \x / \y / \diameter in {0 / 0 / 2mm, 1 / 1 / 3mm, 2 / 0 / 1mm}
    \draw (\x,\y) circle (\diameter);

  % Same effect
  \foreach \center/\diameter in {{(0,0)/2mm}, {(1,1)/3mm}, {(2,0)/1mm}}
    \draw[yshift=2.5cm] \center circle (\diameter);
\end{tikzpicture}
```



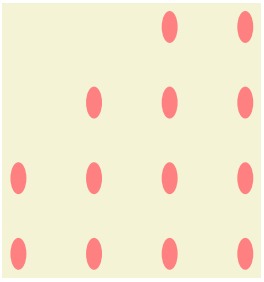

```
\begin{tikzpicture}[cap=round,line width=3pt]
  \filldraw [fill=yellow] (0,0) circle (2cm);

  \foreach \angle / \label in
    {0/3, 30/2, 60/1, 90/12, 120/11, 150/10, 180/9,
     210/8, 240/7, 270/6, 300/5, 330/4}
  {
    \draw[line width=1pt] (\angle:1.8cm) -- (\angle:2cm);
    \draw (\angle:1.4cm) node{\textsf{\label}};
  }

  \foreach \angle in {0,90,180,270}
    \draw[line width=2pt] (\angle:1.6cm) -- (\angle:2cm);

  \draw (0,0) -- (120:0.8cm); % hour
  \draw (0,0) -- (90:1cm);    % minute
\end{tikzpicture}%
```



```
\tikz[shading=ball]
  \foreach \x / \cola in {0/red,1/green,2/blue,3/yellow}
    \foreach \y / \colb in {0/red,1/green,2/blue,3/yellow}
      \shade[ball color=\cola!50!\colb] (\x,\y) circle (0.4cm);
```

\breakforeach

If this command is given inside a `\foreach` command, no executions of the ⟨*commands*⟩ will occur. However, the current execution of the ⟨*commands*⟩ is continued normally, so it is probably best to use this command only at the end of a `\foreach` command.



```
\begin{tikzpicture}
  \foreach \x in {1,...,4}
    \foreach \y in {1,...,4}
    {
      \fill[red!50] (\x,\y) ellipse (3pt/6pt);

      \ifnum \x<\y
        \breakforeach
      \fi
    }
\end{tikzpicture}
```

13 Page Management

This section describes the `pgfpages` packages. Although this package is not concerned with creating pictures, its implementation relies so heavily on PGF that it is documented here. Currently, this pages only works with LaTeX, but if you are advanterous, feel free to hack the code so that is also works with plain TeX.

The aim of `pgfpages` is to provide a flexible way of putting multiple pages on a single page *inside TeX*. Thus, `pgfpages` is quite different from useful tools like `psnup` or `pdfnup` insofar as it creates its output in a single pass. Furthermore, it works uniformly with both `latex` and `pdflatex`, making it easy to put multiple pages on a single page without any fuss.

One word of warning: **using `pgfpages` will destroy hyperlinks**. Actually, the hyperlinks are not destroyed, only they will appear at totally wrong positions on the final output. This is due to a fundamental flaw in the PDF specification: In PDF, the bounding rectangle of a hyperlink is given in "absolute page coordinates" and translations or rotations do not affect them. Thus, the transformations applied by `pgfpages` to put the pages where you want them are (cannot, even) be applied to the coordinates of hyperlinks. It is unlikely that this will change in the forseeable future.

13.1 Basic Usage

The internals of `pgfpages` are complex since the package can do all sorts of interesting tricks. For this reason, so-called *layouts* are predefined that setup all option in appropriate ways.

You use a layout as follows:

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagelayout{2 on 1}[a4paper,landscape,border shrink=5mm]

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The layout `2 on 1` puts two pages on a single page. The option `a4paper` tells `pgfpages` that the *resulting* page (called the *physical* page in the following) should be `a4paper` and it should be landscape (which is quite logical since putting two portrait pages next to each other gives a landscape page). Normally, the *logical* page, that is, the page that TeX "thinks" that it is typesetting, will have the same size, but this need not be the case. `pgfpages` will automatically scale down the logical page such that two logical pages fit next to each other inside a DIN A4 page.

The `border shrink` tells `pgfpages` that it should add an additional 5mm to the shrinking such that a 5mm-wide border is shown around the resulting logical pages.

As a second example, let's put to pages produced by BEAMER on a single page:

```
\documentclass{beamer}

\usepackage{pgfpages}
\pgfpagelayout{2 on 1}[a4paper,border shrink=5mm]

\begin{document}
\begin{frame}
  This text is shown on top.
\end{frame}
\begin{frame}
  This text is shown on the right.
\end{frame}
\end{document}
```

Note that we do not use the `landscape` option since BEAMER's logical pages are already in landscape mode and putting two landscape pages on top of each other results in a portrait page. However, if you had used the `4 on 1` layout, you would have to add the `landscape` once more, using the `8 on 1` you must not, using `16 on 1` you need it yet again. And, no, there is no `32 on 1` layout.

Another word of caution: **using `pgfpages` will produce wrong page numbers in the `.aux` file**. The reason is that TeX instantiates the page numbers when writing an `.aux` file only when the physical page is shipped out. Fortunately, this problem is easy to fix: First, typeset our file normally without using the `\pgfpagelayout` command (just put the comment marker `%` before it) Then, rerun TeX with the

\pgfpagelayout command included and add the command \nofiles. This command ensures that the .aux file is not modified, which is exactly what you want. So, to typeset the above example, you should actually first TEX the following file:

```
\documentclass{article}

\usepackage{pgfpages}
%%\pgfpagelayout{2 on 1}[a4paper,landscape,border shrink=5mm]
%%\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

and then typeset

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagelayout{2 on 1}[a4paper,landscape,border shrink=5mm]
\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The final basic example is the `resize to` layout (it works a bit like a hypothetical `1 on 1` layout). This layout resizes the logical page such that is fits the specified physical size. Since this does not change the page numbering, you need not worry about the .aux files with this layout. For example, adding the following lines will ensure that the physical output will fit on DIN A4 paper:

```
\usepackage{pgfpages}
\pgfpagelayout{resize to}[a4paper]
```

This can be very useful when you have to handle lots of papers that are typeset for, say, letter paper and you have an A4 printer or the other way round. For example, the following article will be fit for printing on letter paper:

```
\documentclass[a4paper]{article}
%% a4 is currently the logical size and also the physical size

\usepackage{pgfpages}
\pgfpagelayout{resize to}[letterpaper]
%% a4 is still the logical size, but letter is the physical one

\begin{document}
  \title{My Great Article}
...
\end{document}
```

13.2 The Predefined Layouts

This section explains the predefined layouts in more detail. You select a layout using the following command:

\pgfpagelayout{⟨*layout*⟩}[⟨*options*⟩]

Installs the specified ⟨*layout*⟩ with the given ⟨*options*⟩ set. The predefined layouts and their permissible options are explained below.

If this function is called multiple times, only the last call "wins." You can thereby overwrite any previous settings. In particular, layouts *do not* accumulate.

Example: `\pgfpagelayout{resize to}[a4paper]`

\pgfpagelayout{resize to}[⟨*options*⟩]

This layout is used to resize every logical page to a specified physical size. To determine the target size, the following options may be given:

- `custom physical paper height=`⟨*size*⟩ sets the hieght of the physical pape size to ⟨*size*⟩.
- `custom physical paper width=`⟨*size*⟩ sets the width of the physical pape size to ⟨*size*⟩.
- `a0paper` sets the physical page size to DIN A0 paper.
- `a1paper` sets the physical page size to DIN A1 paper.
- `a2paper` sets the physical page size to DIN A2 paper.
- `a3paper` sets the physical page size to DIN A3 paper.
- `a4paper` sets the physical page size to DIN A4 paper.
- `a5paper` sets the physical page size to DIN A5 paper.
- `a6paper` sets the physical page size to DIN A6 paper.
- `letterpaper` sets the physical page size to the American letter paper size.
- `legalpaper` sets the physical page size to the American legal paper size.
- `executivepaper` sets the physical page size to the American executive paper size.
- `landscape` swaps the height and the width of the physical paper.
- `border shrink=`⟨*size*⟩ additionally reduces the size of the logical page on the physical page by ⟨*size*⟩.

`\pgfpagelayout{2 on 1}[`⟨*options*⟩`]`

Puts two logical pages alongside each other on each physical page if the logical height is larger than the logical width (logical pages are in portrait mode). Otherwise, two logical pages are put on top of each other (logical pages are in landscape mode). When using this layout, it is advisable to use the `\nofiles` command, but this is not done automatically.

The same ⟨*options*⟩ as for the `resize to` layout an be used, plus the following option:

- `odd numbered pages right` places the first page on the right.

`\pgfpagelayout{4 on 1}[`⟨*options*⟩`]`

Puts four logical pages on a single physical page. The same ⟨*options*⟩ as for the `resize to` layout an be used.

`\pgfpagelayout{8 on 1}[`⟨*options*⟩`]`

Puts eight logical pages on a single physical page. As for `2 on 1`, the orientation depends on whether the logical pages are in landscape mode or in portrait mode.

`\pgfpagelayout{16 on 1}[`⟨*options*⟩`]`

This is for the management...

`\pgfpagelayout{rounded corners}[`⟨*options*⟩`]`

This layout adds "rounded corners" to every page, which, supposedly, looks nicer during presentations with projectors (personallly, I doubt this). This is done by (possibly) resizing the page to the physical page size. Then four black rectangles are drawn in each corner. Next, a clipping region is set up that contains all of the logical page except for little rounded corners. Finally, the logical page is draw, clipped against the clipping region.

Note that every logical page should fill its background for this to work.

In addition to the ⟨*options*⟩ that can be given to `resize to` the following options may be given.

- `corner width=`⟨*size*⟩ specifies the size of the corner.

```
\documentclass{beamer}
\usepackage{pgfpages}
\pgfpagelayout{rounded corners}[corner width=5pt]
\begin{document}
...
\end{document}
```

`\pgfpagelayout{two screens with lagging second}[⟨options⟩]`

This layout puts two logical pages alongside each other. The second page always shows what the main page showed on the previous physical page. Thus, the second page "lags behind" the main page. This can be useful when you have to projectors attached to your computer and can show different parts of a physical page on different projectors.

The following ⟨*options*⟩ may be given:

- `second right` puts the second page right of the main page. This will make the physical pages twice as wide as the logical pages, but it will retain the height.
- `second left` puts the second page left, otherwise it behave the same as `second right`.
- `second bottom` puts the second page below the main page. This make the physical pages twice as high as the logical ones.
- `second top` works like `second bottom`.

`\pgfpagelayout{two screens with optional second}[⟨options⟩]`

This layout works similarly to `two screens with lagging second`. The difference is that the contents of the second screen only changes when one of the commands `\pgfshipoutlogicalpage{2}{`⟨*box*⟩`}` or `\pgfcurrentpagewillbelogicalpage{2}` is called. The first puts the given ⟨*box*⟩ on the second page. The second specifies that the current page should be put there, once it is finished.

The same options as for `two screens with lagging second` may be given.

You can define your own predefined layouts using the following command:

`\pgfdefpagelayout{`⟨*layout*⟩`}{`⟨*before actions*⟩`}{`⟨*after actions*⟩`}`

This command predefines a ⟨*layout*⟩ that can later be installed using the `\pgfpagelayout` command.

When `\pgfpagelayout{`⟨*layout*⟩`}[`⟨*options*⟩`]` is called, the following happens: First, the ⟨*before actions*⟩ are executed. They can be used, for example, to setup default values for keys. Next, `\setkeys{pgfpagelayoutoption}{`⟨*options*⟩`}` is executed. Finally, the ⟨*after actions*⟩ are executed.

```
\pgfdefpagelayout{resize to}
{
  \def\pgfpageoptionborder{0pt}
}
{
  \pgfpagesoptions
  {%
    logical pages=1,%
    physical height=\pgfpageoptionheight,%
    physical width=\pgfpageoptionwidth%
  }
  \pgfsetuppage{1}
  {%
    resized width=\pgfphysicalwidth,%
    resized height=\pgfphysicalheight,%
    border shrink=\pgfpageoptionborder,%
    center=\pgfpoint{.5\pgfphysicalwidth}{.5\pgfphysicalheight}%
  }%
}
```

13.3 Defining a Layout

If none of the predefined layouts meets your problem or if you wish to modify them, you can create layouts from scratch. This section explains how this is done.

Basically, `pgfpages` hooks into TEX's `\shipout` function. This function is called whenever TEX has completed typesetting a page and wishes to send this page to the `.dvi` or `.pdf` file. Now, `pgfpage` redefines this command. Instead of sending the page to the output file, `pgfpages` stores it in an internal box and then acts as if the page had been output. When TEX tries to output the next page using `\shipout`, this call is once more intercepted and the page is stored in another box. These boxes are called *logical pages*.

At some point, enough logical pages have been accumulated such that a *physical page* can be output. When this happens, `pgfpages` possibly scales, rotates, and translates the logical pages (and possibly even

does further modifications) and then puts them at certain positions of the *physical* page. Once this page is fully assembled, the "real" or "original" `\shipout` is called to send the physical page to the output file.

In reality, things are slightly more complicated. First, once a physical page has been shipped out, the logical pages are usually voided, but this need not be the case. Instead, it is possible that certain logical page just retain their contents after the physical page has been shipped out and these pages need not be filled once more before a physical shipout can occur. However, the contents of these logical pages can still be changed using special commands. It is also possible that after a shipout certain logical pages are filled with the contents of *other* logical pages.

A *layout* defines for each logical page where it will go on the physical page and which further modifications should be done. The following two commands are used to define the layout:

`\pgfpageoptions{`⟨*options*⟩`}`

This command sets the "global" page options. For example, it is used to specify how many logical pages there are and how many logical pages must be accumulated before a physical page is shipped out. How each individual logical page is tyhpeset is specified using the command `\pgfsetuppage`, described later.

Example: A layout for putting two portrait pages on a single landscape page:

```
\pgfpagesoptions
{%
  logical pages=2,%
  physical height=\paperwidth,%
  physical width=\paperheight,%
}

\pgfsetuppage{1}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
\pgfsetuppage{2}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.75\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
```

The following ⟨*options*⟩ may be set:

- `logical pages=`⟨*logical pages*⟩ specified how many logical pages there are, in total. These are numbered 1 to ⟨*logical pages*⟩.
- `first logical shipout=`⟨*first*⟩. See the the next option. By default, ⟨*first*⟩ is 1.
- `last logical shipout=`⟨*last*⟩. Together with the previous option, these two options define an interval of pages inside the range 1 to ⟨*logical pages*⟩. Only this range is used to store the pages that are shipped out by TEX. This means that after a physical shipout has just occured (or at the beginning), the first time TEX wishes to perform a shipout, the page to be shipped out is stored in logical page ⟨*first*⟩. The next time TEX performs a shipout, the page is sotred in logical page ⟨*first*⟩ + 1 and so on, until the logical page ⟨*last*⟩ is also filled. Once this happens, a physcial shipout occurs and the process starts once more.

  Note that logical pages that lie outside the interval between ⟨*first*⟩ and ⟨*last*⟩ are filled only indirectly or when special commands are used.

  By default, ⟨*last*⟩ equals ⟨*logical pages*⟩.
- `current logical shipout=`⟨*current*⟩ changes an internal counter such that TEX's next logical shipout will be stored in logical page ⟨*current*⟩.

  This optino can be used to "warp" the logical page filling mechanism to a certain page. You can both skip logical pages and overwrite already filled logical pages. After the logical page ⟨*current*⟩ has been filled, the internal counter is incremented normally as if the logical page ⟨*current*⟩ had been "reached" normally. If you specify a ⟨*current*⟩ larger to ⟨*last*⟩, a physical shipout will occur after the logical page ⟨*current*⟩ has been filled.
- `physical height=`⟨*height*⟩ specifies the height of the physical pages. This height is typically different from the normal `\paperheight`, which is used by TEX for its typesetting and page breaking purposes.

- `physical width=`⟨*width*⟩ specifies the physical width.

`\pgfsetuppage{`⟨*logical page number*⟩`}{`⟨*options*⟩`}`

This command is used to specify where the logical page number ⟨*logical page number*⟩ will be placed on the physical page. In addtion, this command can be used to install additional "code" to be executed when this page is put on the physical page.

The number ⟨*logical page number*⟩ should be between 1 and ⟨*logical pages*⟩, which has previously been installed using the `\pgfpageoptions` command.

The following ⟨*options*⟩ may be given:

- `center=`⟨*pgf point*⟩ specifies the center of the logical page inside the physical page as a PGF-point. The origin of the coordinate system of the physical page is at the *lower* left corner.

```
\pgfsetuppage{1}
{% center logical page on middle of left side
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
}
```

- `resized width=`⟨*size*⟩ specifies the width that the logical page should have *at most* on the physical page. To achieve this width, the pages is scaled down appropriately *or more*. The "or more" part can happen if the `resize height` option is also used. In this case, the scaling is chosen such that both the specified height and width are met. The aspect ratio of a logical page is not modified.
- `resized height=`⟨*height*⟩ specifies the maxmimum height of the logical page.
- `original width=`⟨*width*⟩ specifies the width the TEX "thinks" that the logical page has. This width is `\paperwidth` at the point of invocation, by default. Note that setting this width to something different from `\paperwidth` does *not* change the `\pagewidth` during TEX's typesetting. You have to do that yourself.

  You need this option only for special logical pages that have a height or width different from the normal one and for which you will (later on) set these sizes yourself.
- `original height=`⟨*height*⟩ works like `origianl width`.
- `scale=`⟨*factor*⟩ scales the page by at least the given ⟨*factor*⟩. A ⟨*factor*⟩ of `0.5` will half the size of the page, a factor or `2` will double the size. "At least" means that if options like `resize height` are given and if the scaling required to meet that option is less than ⟨*factor*⟩, that other scaling is used instead.
- `scalex=`⟨*factor*⟩ scales the logical page along the x-axis by the given ⟨*factor*⟩. This scaling is done independently of any other scaling. Mostly, this option is useful for a factor of `-1`, which flips the page along the y-axis. The aspect ratio is not kept.
- `scaley=`⟨*factor*⟩ works like `scalex`, only for the y-axis.
- `rotation=`⟨*degree*⟩ rotates the page by ⟨*degree*⟩ around its center. Use a degree of `90` or `-90` to go from portait to landscape and back. The rotation need not be a multiple of `90`.
- `copy from=`⟨*logical page number*⟩. Normally, after a physical shipout has occured, all logical pages are voided in a loop. However, if this option is given, the current logical page is filled with the contents of the old logical page number ⟨*logical page number*⟩.

  Example: Have logical page 2 retain its contents:

```
\pgfsetuppage{2}{copy from=2}
```

  Example: Let logical page 2 show what logical page 1 showed on the just-shipped-out physical page:

```
\pgfsetuppage{1}{copy from=1} % necessary so that page 1 does get
                              % voided before it is copied to page 2
\pgfsetuppage{2}{copy from=1}
```

- `border shrink=`⟨*size*⟩ specifies an addition reduction of the size to which the page is page is scaled down.

- `border code=`⟨*code*⟩. When this option is given, the ⟨*code*⟩ is executed before the page box is inserted with a path preinstalled that is a rectangle around the current logical page. Thus, setting ⟨*code*⟩ to `\pgfstroke` draws a rectangle around the logical page. Setting ⟨*code*⟩ to `\pgfsetlinewidth{3pt}\pgfstroke` results in a thick (ugly) frame. Adding dashes and filling can result in arbitrarily funky and distracting borders.

  You can also call `\pgfdiscardpath` and add your own path construction code (for example to paint a rectangle with rounded corners). The coordinate system is setup in such a way that a rectangle starting at the origin and having the height and width of TEX-box 0 will result in a rectangle filling exactly the logical page currently being put on the physical page. The logical page is inserted *after* these commands have been executed.

  Example: Add a rectangle around the page:

  ```
  \pgfsetuppage{1}{border code=\pgfstroke}
  ```

- `corner width=`⟨*size*⟩ adds black "rounded corners" to the page. See the description of the predefined layout `rounded corners` on page 98.

14 Extended Color Support

This section documents the package `xxcolor`, which is currently distributed as part of PGF. This package extends the `xcolor` package, written by Uwe Kern, which in turn extends the `color` package. I hope that the commands in `xxcolor` will some day migrate to `xcolor`, such that this package becomes superfluous.

The main aim of the `xxcolor` package is to provide an environment inside which all colors are "washed out" or "dimmed." This is useful in numerous situations and must typically be achieved in a roundabout manner if such an environment is not available.

`\begin{colormixin}{`⟨*mix-in specification*⟩`}`
⟨*environment contents*⟩
`\end{colormixin}`

The mix-in specification is applied to all colors inside the environment. At the beginning of the environment, the mix-in is applied to the current color, i. e., the color that was in effect before the environment started. A mix-in specification is a number between 0 and 100 followed by an exclamation mark and a color name. When a `\color` command is encountered inside a mix-in environment, the number states what percentage of the desired color should be used. The rest is "filled up" with the color given in the mix-in specification. Thus, a mix-in specification like `90!blue` will mix in 10% of blue into everything, whereas `25!white` will make everything nearly white.

Red text,washed-out red text, washed-out blue text, dark washed-out blue text, dark washed-out green text, back to washed-out blue text,and back to red.

```
\begin{minipage}{3.5cm}\raggedright
\color{red}Red text,%
\begin{colormixin}{25!white}
  washed-out red text,
  \color{blue} washed-out blue text,
  \begin{colormixin}{25!black}
    dark washed-out blue text,
    \color{green} dark washed-out green text,%
  \end{colormixin}
  back to washed-out blue text,%
\end{colormixin}
and back to red.
\end{minipage}%
```

Note that the environment only changes colors that have been installed using the standard LaTeX `\color` command. In particular, the colors in images are not changed. There is, however, some support offered by the commands `\pgfuseimage` and `\pgfuseshading`. If the first command is invoked inside a `colormixin` environment with the parameter, say, `50!black` on an image with the name `foo`, the command will first check whether there is also a defined image with the name `foo.!50!black`. If so, this image is used instead. This allows you to provide a different image for this case. If you nest `colormixin` environments, the different mix-ins are all appended. For example, inside the inner environment of the above example, `\pgfuseimage{foo}` would first check whether there exists an image named `foo.!50!white!25!black`.

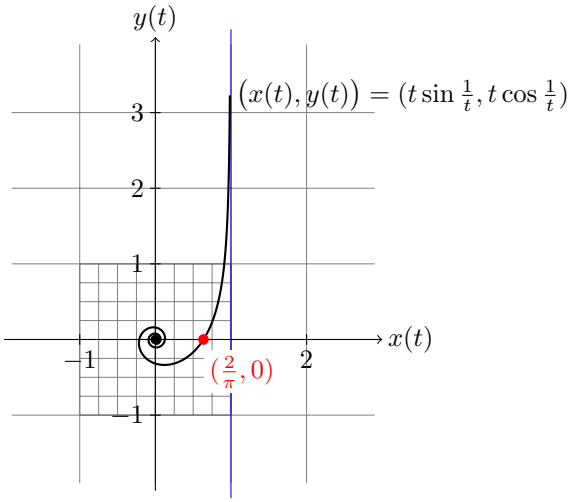
`\colorcurrentmixin`

Expands to the current accumulated mix-in. Each nesting of a `colormixin` adds a mix-in to this list.

!75!white should be "75!white"
!75!black!75!white should be "75!black!75!white"
!50!white!75!black!75!white should be "50!white!75!black!75!white"

```
\begin{minipage}{\linewidth-6pt}\raggedright
\begin{colormixin}{75!white}
  \colorcurrentmixin\ should be ``75!white''\par
  \begin{colormixin}{75!black}
    \colorcurrentmixin\ should be ``75!black!75!white''\par
    \begin{colormixin}{50!white}
      \colorcurrentmixin\ should be ``50!white!75!black!75!white''\par
    \end{colormixin}
  \end{colormixin}
\end{colormixin}
\end{minipage}
```

Part IV
The Basic Layer




```
\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] $(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))$
    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}
```

15 Design Principles

This section describes the basic layer of PGF. This layer is build on top of the system layer. Whereas the system layer just provides the absolute minimum for drawing graphics, the basic layer provides numerous commands that make it possible to create sophisticated graphics easily and also quickly.

The basic layer does not provide a convenient syntax for describing graphics, which is left to frontends like Ti*k*Z. For this reason, the basic layer is typically used only by "other programs." For example, the BEAMER package used the basic layer extensively, but does not need a convenient input syntax. Rather, speed and flexibility are needed when BEAMER creates graphics.

The following basic design principles underlie the basic layer:

1. Structuring into a core and several optional packages.
2. Consistently named TEX macros for all graphics commands.
3. Path-centered description of graphics.
4. Coordinate transformation system.

15.1 Core and Optional Packages

The basic layer consists of a *core package*, called `pgfcore`, which provides the most basic commands, and several optional package like `pgfbaseshade` that offer more special-purpose commands.

You can include the core by saying `\usepackage{pgfcore}` or, as a plain TEX user, `\input pgfcore.tex`.

The following optional packages are provided by the basic layer:

- `pgfbaseplot` provides commands for plotting functions.
- `pgfbaseshapes` provides commands for drawing shapes and nodes.
- `pgfbaseimage` provides commands for including external images. The `graphicx` package does a much better job at this than the `pgfbaseimage` package does, so you should normally use `\includegraphics` and not `\pgfimage`. However, in some situations (like when masking is needed or when plain TEX is used) this package is needed.

If you say `\usepackage{pgf}` or `\input pgf.tex`, all of the optional packages are loaded (as well as the core and the system layer).

15.2 Communicating with the Basic Layer via Macros

In order to "communicate" with the basic layer you use long sequences of commands that start with `\pgf`. You are only allowed to give these commands inside a `{pgfpicture}` environment. However, it is possible to "do other things" between the commands. For example, you might use one command to move to a certain point, then have a complicated computation of the next point, and then move there.

```
\newdimen\myypos
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{\myypos}}
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \advance \myypos by 1cm
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following naming conventions are used in the basic layer:

1. All commands and environments start with `pgf`.
2. All commands that specify a point (a coordinate) start with `\pgfpoint`.
3. All commands the extend the current path start with `\pgfpath`.
4. All commands that set/change a graphics parameter start with `\pgfset`.
5. All commands that use a previously declared object (like a path, image or shading) start with `\pgfuse`.

6. All commands having to do with coordinate transformations start with `\pgftransform`.
7. All commands having to do with declaring arrows start with `\pgfarrows`.
8. All commands for "quickly" extending or drawing a path start with `\pgfpathq` or `\pgfusepathq`.

15.3 Path-Centered Approach

In PGF, the most important entity is the *path*. All graphics are composed of numerous paths that can be stroked, filled, shaded, or clipped against. Paths can be closed or open, they can self-intersect and consist of unconnected parts.

Paths are first *constructed* and then *used*. In order to construct a path, you can use commands starting with `\pgfpath`. Each time such a command is called, the current path is extended in some way.

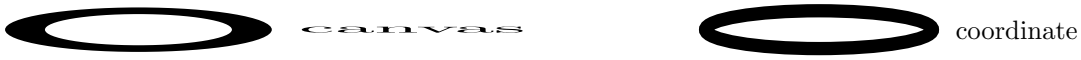
Once a path has been completely constructed, you can use it using the command `\pgfusepath`. Depending on the parameters given to this command, the path will be stroked (drawn) or filled or subsequent drawings will be clipped against this path.

15.4 Coordinate Versus Canvas Transformations

PGF provides two transformation systems: PGF's own *coordinate* transformation matrix and PDF or PostScript's *canvas* transformation matrix. These two systems are quite different. Whereas a scaling by a factor of, say, 2 of the canvas causes *everything* to be scaled by this factor (including the thickness of lines and text), a scaling of two in the coordinate system causes only the *coordinates* to be scaled, but not the line width nor text.

By default, all transformations only apply to the coordinate transformation system. However, using the command `\pgflowlevel` it is possible to apply a transformation to the canvas.

Coordinate transformations are often preferable over canvas transformations. Text and lines that are transformed using canvas transformations suffer from differing sizes and lines whose thickness differs depending on whether the line is horizontal or vertical. To appreciate the difference, consider the following two "circles" both of which have been scaled in the x-direction by a factor of 3 and by a factor of 0.5 in the y-direction. The left circle uses a canvas transformation, the right uses PGF's coordinate transformation (some viewers will render the left graphic incorrectly since they do no apply the low-level transformation the way they should):



16 Specifying Coordinates

16.1 Overview

Most PGF commands expect you to provide the coordinates of a *point* or *coordinate* inside your picture. Points are always "local" to your picture, that is, they never refer to an absolute position on the page, but to a position inside the current `{pgfpicture}` environment.

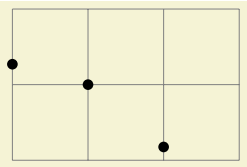
To specify a coordinate, you can use commands that start with `\pgfpoint`. The simplest of these commands is just `\pgfpoint` and it takes two arguments: An x-dimension and a y-dimension. These dimensions are given as TEX-dimensions.

16.2 Basic Coordinate Commands

The following commands are the most basic for specifying a coordinate.

`\pgfpoint{`⟨*x coordinate*⟩`}{`⟨*y coordinate*⟩`}`

Yields a point location. The coordinates are given as TEX dimensions.



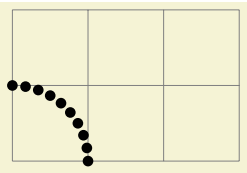
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfpathcircle{\pgfpoint{1cm}{1cm}} {2pt}
  \pgfpathcircle{\pgfpoint{2cm}{5pt}} {2pt}
  \pgfpathcircle{\pgfpoint{0pt}{.5in}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

`\pgfpointorigin`

Yields the origin. Same as `\pgfpoint{0pt}{0pt}`.

`\pgfpointpolar{`⟨*degree*⟩`}{`⟨*radius*⟩`}`

Yields a point location given in polar coordinates. You can specify the angle only in degrees, radians are not supported.



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

  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

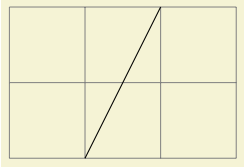
16.3 Coordinates in the xy- and xyz-Coordinate Systems

Coordinates can also be specified as multiples of an x-vector and a y-vector. Normally, the x-vector points one centimeter in the x-direction and the y-vector points one centimeter in the y-direction, but using the commands `\pgfsetxvec` and `\pgfsetyvec` they can be changed. Note that the x- and y-vector do not necessarily point "horizontally" and "vertically."

It is also possible to specify a point as a multiple of three vectors, the x-, y-, and z-vector. This is useful for creating simple three dimensional graphics.

`\pgfpointxy{`⟨s_x⟩`}{`⟨s_y⟩`}`

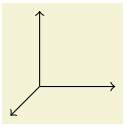
Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector plus s_z times the z-vector.

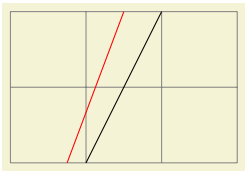


```
\begin{pgfpicture}
  \pgfsetarrowsend{to}

  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{0}{1}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{1}{0}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{1}{0}{0}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetxvec{⟨*point*⟩}

Sets that current x-vector for usage in the xyz-coordinate system.

Example:



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

  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}

  \color{red}
  \pgfsetxvec{\pgfpoint{0.75cm}{0cm}}
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetyvec{⟨*point*⟩}

Works like \pgfsetyvec.

\pgfsetzvec{⟨*point*⟩}

Works like \pgfsetzvec.

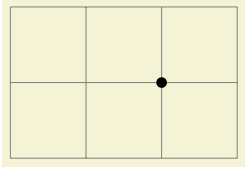
16.4 Building Coordinates From Other Coordinates

Many commands allow you to construct a coordinate in terms of other coordinates.

16.4.1 Basic Manipulations of Coordiantes

\pgfpointadd{⟨v_1⟩}{⟨v_2⟩}

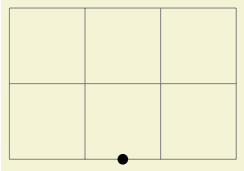
Returns the sum vector $\langle v_1\rangle + \langle v_2\rangle$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointadd{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointscale{⟨*factor*⟩}{⟨*coordinate*⟩}

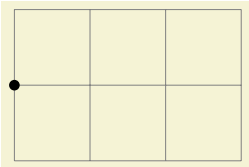
Returns the vector ⟨*factor*⟩⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointscale{1.5}{\pgfpoint{1cm}{0cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointdiff{⟨*start*⟩}{⟨*end*⟩}

Returns the difference vector ⟨*end*⟩ − ⟨*start*⟩.

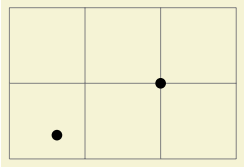


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointdiff{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointnormalised{⟨*point*⟩}

This command returns a normalised version of ⟨*point*⟩, that is, a vector of length 1pt pointing in the direction of ⟨*point*⟩. If ⟨*point*⟩ is the 0-vector or extremely short, a vector of length 1pt pointing upwards is returned.

This command is *not* implemented by calculating the length of the vector, but rather by calculating the angle of the vector and then using (something equivalent to) the **\pgfpointpolar** command. On the one hand this ensures that the point will really have length 1pt, on the other hand it is not guaranteed that the vector will *precisely* point in the direction of ⟨*point*⟩ due to the fact that the polar tables are accurate only up to one degree. Normally, this is not a problem.



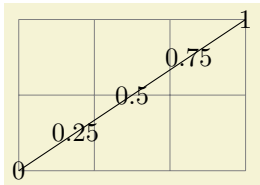
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointscale{20}
    {\pgfpointnormalised{\pgfpoint{2cm}{1cm}}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

16.4.2 Points Travelling along Lines and Curves

The commands in this section allow you to specify points on a line or a curve. Imaging a point "travelling" along a curve from some point p to another point q. At time $t = 0$ the point is at p and at time $t = 1$ it is at q and at time, say, $t = 1/2$ it is "somewhere in the middle." The exact location at time $t = 1/2$ will not necessarily be the "halfway point," that is, the point whose distance on the curve from p and q is equal. Rather, the exact location will depend on the "speed" at which the point is travelling, which in turn depends on the lengths of the support vectors in a complicated manner. If you are interested in the details, please see a good book on Beziér curves.

\pgfpointlineattime{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point q*⟩}

Yields a point that is the tth fraction between p and q, that is, $p + t(q - p)$. For $t = 1/2$ this is the middle of p and q.

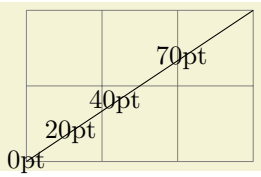



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=
      \pgfpointlineattime{\t}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]{\t}}
\end{tikzpicture}
```

\pgfpointlineatdistance{⟨*distance*⟩}{⟨*start point*⟩}{⟨*end point*⟩}

Yields a point that is located ⟨*distance*⟩ many units removed from the start point in the direction of the end point. In other words, this is the point that results if we travel ⟨*distance*⟩ steps from ⟨*start point*⟩ towards ⟨*end point*⟩.

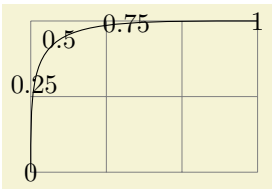
Example:




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {0pt,20pt,40pt,70pt}
    {\pgftext[at=
      \pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]{\d}}
\end{tikzpicture}
```

\pgfpointcurveattime{⟨*time* t⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

Yields a point that is on the Beziér curve from p to q with the support points s_1 and s_2. The time t is used to determine the location, where $t = 0$ yields p and $t = 1$ yields q.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=\pgfpointcurveattime{\t}{\pgfpointorigin}
                                        {\pgfpoint{0cm}{2cm}}
                                        {\pgfpoint{0cm}{2cm}}
                                        {\pgfpoint{3cm}{2cm}}]{\t}}
\end{tikzpicture}
```

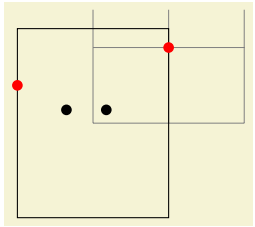
16.4.3 Points on Borders of Objects

The following commands are useful to specify a point that lies on the border of special shapes. They are used, for example, by the shape mechanism to determine border points of shapes.

\pgfpointborderrectangle{⟨*direction point*⟩}{⟨*corner*⟩}

This command returns a point that lies on the intersection of a line starting at the origin and going towards the point ⟨*direction point*⟩ and a rectangle whose center is in the origin and whose upper right corner is at ⟨*corner*⟩.

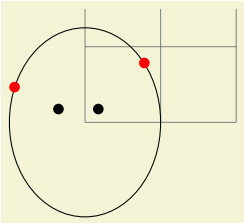
The ⟨*direction point*⟩ should have length "about 1pt," but it will be normalised automatically. Nevertheless, the "nearer" the length is to 1pt, the less rounding errors there will be.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathrectanglecorners{\pgfpoint{-1cm}{-1.25cm}}{\pgfpoint{1cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointborderellipse{⟨*direction point*⟩}{⟨*corner*⟩}

This command works like the corresponding command for rectangles, only this time the ⟨*corner*⟩ is the corner of the bounding rectangle of an ellipse.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathellipse{\pgfpointorigin}{\pgfpoint{1cm}{0cm}}{\pgfpoint{0cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

16.5 Extracting Coordinates

There are two commands that can be used to "extract" the x- or y-coordinate of a coordiante. The extraction process will "un-big-point-correct" the coordinates (if you do not know what this means, things will work fine).

\pgfextractx{⟨*dimension*⟩}{⟨*point*⟩}

Sets the TEX-⟨*dimension*⟩ to the x-coordinate of the point.

```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
%% \mydim is now 2cm
```

\pgfextracty{⟨*dimension*⟩}{⟨*point*⟩}

Like \pgfextractx, except for the y-coordinate.

16.6 Internals of How Point Commands Work

As a normal user of PGF, you do not need to read this section. It is relevant only if you need to understand how the point commands work internally.

When a command like \pgfpoint{1cm}{2pt} is called, all that happens is that the two TEX-dimension variables \pgf@x and \pgf@y are set to 1cm and 2pt, respectively. A command like \pgfpathmoveto that takes a coordinate as parameter will just execute this parameter and then use the values of \pgf@x and \pgf@y as the coordinates to which it will move the pen on the current path.

Commands like \pgfpointnormalised modify other variables besides \pgf@x and \pgf@y during the computation of the final values of \pgf@x and \pgf@y, it is a good idea to enclose a call of a command like \pgfpoint in a TEX-scope and then make the changes of \pgf@x and \pgf@y global as in the following example:

```
...
{ % open scope
  \pgfpointnormalised{\pgfpoint{1cm}{1cm}}
  \global\pgf@x=\pgf@x % make the change of \pgf@x persist past the scope
  \global\pgf@y=\pgf@y % make the change of \pgf@y persist past the scope
}
% \pgf@x and \pgf@y are now set correctly, all other variables are
% unchanged
```

Since this situation arises very often, the macro \pgf@process can be used to perform the above code:

\pgf@process{⟨*code*⟩}

Executes the ⟨*code*⟩ in a scope and then makes \pgf@x and \pgf@y global.

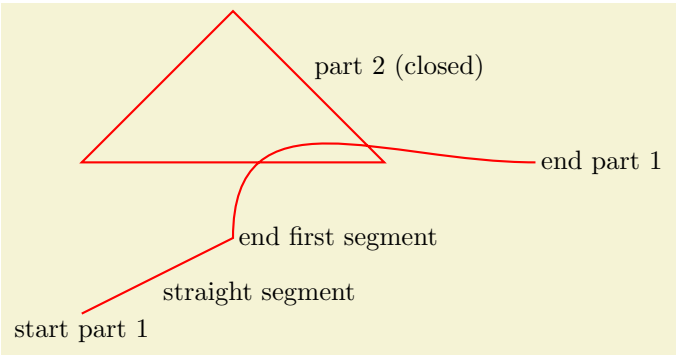
Note that this macro is used often internally. For this reason, it is not a good idea to keep anything important in the variables `\pgf@x` and `\pgf@y` since they will be overwritten and changed frequently. Instead, intermediate values can ge stored in the TEX-dimensions `\pgf@xa`, `\pgf@xb`, `\pgf@xc` and their `y`-counterparts `\pgf@ya`, `\pgf@yb`, `pgf@yc`. For examples, here is the code of the command `\pgfpointadd`:

```
\def\pgfpointadd#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \advance\pgf@x by\pgf@xa%
  \advance\pgf@y by\pgf@ya}
```

17 Constructing Paths

17.1 Overview

The "basic entity of drawing" in PGF is the *path*. A path consists of several parts, each of which is either a closed or open curve. An open curve has a starting point and an end point and inbetween it consists of several *segmets*, each of which is either a straight line or a Beziér curve. Here is an example of a path (in red) consisting of two parts, one open, one closed:




```
\begin{tikzpicture}[scale=2]
  \draw[thick,red]
       (0,0) coordinate (a)
    -- coordinate (ab) (1,.5) coordinate (b)
    .. coordinate (bc) controls +(up:1cm) and +(left:1cm) .. (3,1)  coordinate (c)
       (0,1) -- (2,1) -- coordinate (x) (1,2) -- cycle;

  \draw (a)  node[below] {start part 1}
        (ab) node[below right] {straight segment}
        (b)  node[right] {end first segment}
        (c)  node[right] {end part 1}
        (x)  node[above right]  {part 2 (closed)};
\end{tikzpicture}
```

A path, by itself, has no "effect," that is, it does not leave any marks on the page. It is just a set of points on the plane. However, you can *use* a path in different ways. The most natural actions are *stroking* (also known as *drawing*) and *filling*. Stroking can be imagined as picking up a pen of a certain diameter and "moving it along the path." Filling means that everything "inside" the path is filled with a uniform color. Naturally, the open parts of a path must first be closed before a path can be filled.

In PGF, there are numerous commands for constructing paths, all of which start with `\pgfpath`. There are also commands for *using* paths, though most operations can be performed by calling `\pgfusepath` with an appropriate parameter.

As a side-effect, the path construction commands keep track of two bounding boxes. One is the bounding box for the current path, the other is a bounding box for all paths in the current picture. See Section 17.13 for more details.

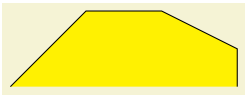
Each path construction command extends the current path in some way. The "current path" is a global entity that persists accross TEX groups. Thus, between calls to the path construction commands you can perform arbitrary computations and even open and closed TEX groups. The current path only gets "flushed" when the `\pgfusepath` command is called (or when the soft-path subsystem is used directly, see Section 29).

17.2 The Move-To Path Operation

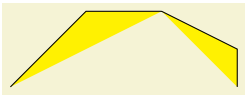
The most basic operation is the move-to operation. It must be given at the beginning of every path. This operation can also be used to start a new part of a path.

`\pgfpathmoveto{`⟨*coordinate*⟩`}`

This command expects a PGF-coordinate like `\pgfpointorigin` as its parameter. When the current path is empty, this operation will start the path at the given ⟨*coordinate*⟩. If a path has already been partly constructed, this command will end the current part of the path and start a new one.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}} % New part
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

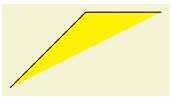
The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

The command will update the bounding box of the current path and picture, if necessary.

17.3 The Line-To Path Operation

\pgfpathlineto{⟨*coordinate*⟩}

This command extends the current path in a straight line to the given ⟨*coordinate*⟩. If this command is given at the beginning of path without any other path construction command given before (in particular without a move-to operation), the TEX file may compile without an error message, but a viewer application may display an error message when trying to render the picture.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{yellow}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

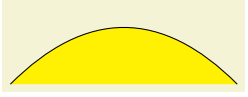
The command will update the bounding box of the current path and picture, if necessary.

17.4 The Curve-To Path Operation

\pgfpathcurveto{⟨*support 1*⟩}{⟨*support 2*⟩}{⟨*coordinate*⟩}

This command extends the current path with a Beziér curve from the last point of the path to ⟨*coordinate*⟩. The ⟨*support 1*⟩ and ⟨*support 2*⟩ are the first and second support point of the Beziér curve. For more information on Beziér curve, please consult a standard textbook on computer graphics.

Like the line-to command, this command may not be the first path construction command in a path.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{1cm}{1cm}}{\pgfpoint{2cm}{1cm}}{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

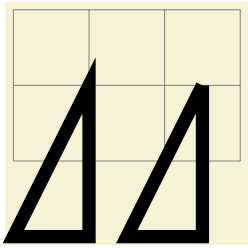
The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

The command will update the bounding box of the current path and picture, if necessary. However, the bounding box is simply made large enough such that it encompasses all of the support points and the ⟨*coordinate*⟩. This will guarantee that the curve is completely inside the bounding box, but the bounding box will typically be quite a bit too large. It is not clear (to me) how this can be avoided without resorting to "some serious math" in order to calculate a precise bounding box.

17.5 The Close Path Operation

\pgfpathclose

This command closed the current part of the path by appending a straight line to the start point of the current part. Note that there *is* a difference between closing a path and using the line-to operation to add a straight line to the start of the current path. The difference is demonstrated by the upper corners of the triangles in the following example:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{5pt}
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{0cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{-1cm}}
  \pgfpathclose
  \pgfpathmoveto{\pgfpoint{2.5cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

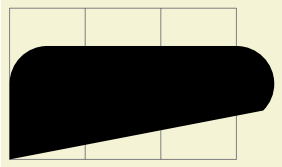
17.6 Arc, Ellipse and Circle Path Operations

The path construction commands that we have discussed up to now are sufficient to create all paths that can be created "at all." However, it is useful to have special commands to create certain shapes, like circle, that arise often in practice.

In the following, the commands for adding (parts of) (transformed) circles to a path.

\pgfpatharc{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*radius*⟩}

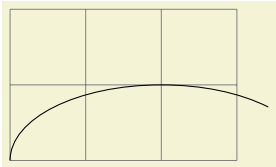
This command appends a part of a circle (or an ellipse) to the current path. Imaging the curve between ⟨*start angle*⟩ and ⟨*end angle*⟩ on a circle of radius ⟨*radius*⟩ (if ⟨*start angle*⟩ < ⟨*end angle*⟩, the curve goes around the circle counterclockwise, otherwise clockwise). This curve is now moved such that the point where the curve starts is the last point of the path. Note that this command will *not* start a new part of the path, which is important for example for filling purposes.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfpatharc{180}{90}{.5cm}
  \pgfpathlineto{\pgfpoint{3cm}{1.5cm}}
  \pgfpatharc{90}{-45}{.5cm}
  \pgfusepath{fill}
\end{tikzpicture}
```

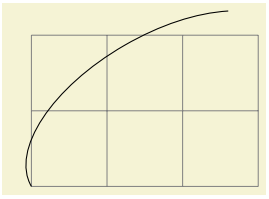
Saying `\pgfpatharc{0}{360}{1cm}` "nearly" gives you a full circle. The "nearly" refers to the fact that the circle will not be closed. You can close it using `\pgfpathclose`.

The ⟨*radius*⟩ need not always be a single TEX dimension. Instead, it can also contain a slash, in which case it must consist of two dimensions separated by this slash. In this case, the first dimension is the x-radius and the second the y-radius of the ellipse from which the curve is taken:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm/1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The axes of the circle or ellipse from which the arc is "taken" always point up and right. However, the current coordinate transformation matrix will have an effect on the arc. This can be used to, say, rotate an arc:

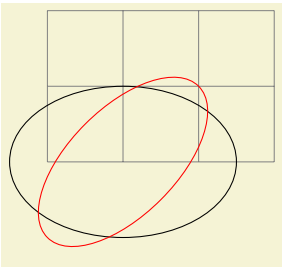


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm/1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will update the bounding box of the current path and picture, if necessary. Unless rotaion or shearing transformations are applied, the bounding box will be tight.

\pgfpathellipse{⟨*center*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

The effect of this command is to append an ellipse to the current path (if the path is not empty, a new part is started). The ellipse's center will be ⟨*center*⟩ and ⟨*first axis*⟩ and ⟨*second axis*⟩ are the axis *vectors*. The same effect as this command can also be achieved using an appropriate sequence of move-to, arc, and close operations, but this command is easier and faster.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1.5cm}{0cm}}
                 {\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
  \color{red}
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{-0.5cm}{0.5cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations to all coordinates of the ellipse. However, the coordinate transformations are applied only after the ellipse is "finished conceptually." Thus, a transformation of 1cm to the right will simply shift the ellipse one centimeter to the right; it will not add 1cm to the x-coordinates of the two axis vectors.

The command will update the bounding box of the current path and picture, if necessary.

\pgfpathcirlce{⟨*center*⟩}{⟨*radius*⟩}

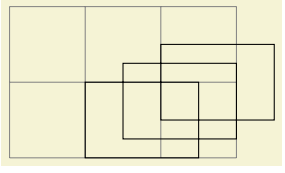
A shorthand for \pgfpathellipse applied to ⟨*center*⟩ and the two axis vectors $(\langle radius\rangle, 0)$ and $(0, \langle radius\rangle)$.

17.7 Rectangle Path Operations

Another shape that arises frequently is the rectangle. Two commands can be used to add a rectangle to the current path. Both commands will start a new part of the path.

\pgfpathrectangle{⟨*corner*⟩}{⟨*diagonal vector*⟩}

Adds a rectangle to the path whose one corner is ⟨*corner*⟩ and whose opposite corner is given by ⟨*corner*⟩ + ⟨*diagonal vector*⟩.

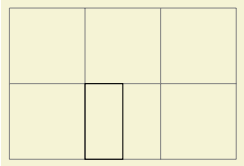


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectangle{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1.5cm}{0.25cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{2cm}{0.5cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

\pgfpathrectanglecorners{⟨*corner*⟩}{⟨*opposite corner*⟩}

Adds a rectangle to the path whose two opposing corners are ⟨*corner*⟩ and ⟨*opposite corner*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectanglecorners{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

17.8 The Grid Path Operation

\pgfpathgrid[⟨*options*⟩]{⟨*lower left*⟩}{⟨*upper right*⟩}

Appends a grid to the current path. That is, a (possibly large) numer of parts are added to the path, each part consisting of a single horizontal or vertical straight line segment.

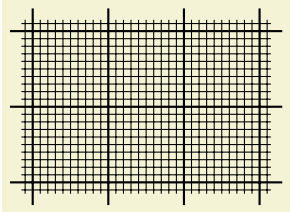
Conceptually, the origin is part of the grid and the grid is clipped to the rectanlge specified by the ⟨*lower left*⟩ and the ⟨*upper right*⟩ corner. However, no clipping occurs (this command just adds parts to the current path). Rather, the points where the lines enter and leave the "clipping area" are computed and used to add simple lines to the current path.

Allowed ⟨*options*⟩ are:

stepx=⟨***dimension***⟩ Sets the horizontal steping to ⟨*dimension*⟩. Default is 1cm.

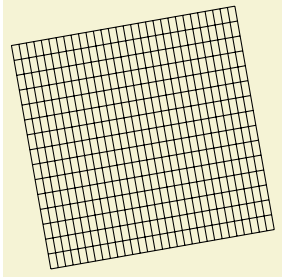
stepy=⟨***dimension***⟩ Sets the vertical steping to ⟨*dimension*⟩. Default is 1cm.

step=⟨***vector***⟩ Sets the horizontal stepping to the x-coordinate of ⟨*vector*⟩ and the vertical steping its y-coordinate.



```
\begin{pgfpicture}
  \pgfsetlinewidth{0.8pt}
  \pgfpathgrid[step={\pgfpoint{1cm}{1cm}}]
    {\pgfpoint{-3mm}{-3mm}}{\pgfpoint{33mm}{23mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{0.4pt}
  \pgfpathgrid[stepx=1mm,stepy=1mm]
    {\pgfpoint{-1.5mm}{-1.5mm}}{\pgfpoint{31.5mm}{21.5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly. As for ellipses, the transformations are applied to the "conceptually finished" grid.



```
\begin{pgfpicture}
  \pgftransformrotate{10}
  \pgfpathgrid[stepx=1mm,stepy=2mm]{\pgfpoint{0mm}{0mm}}{\pgfpoint{30mm}{30mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

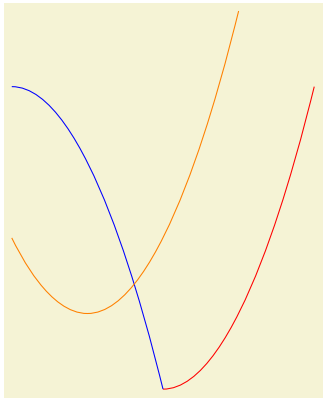
17.9 The Parabola Path Operation

\pgfpathparabola{⟨*bend vector*⟩}{⟨*end vector*⟩}

This command appends two half-parabolas to the current path. The first starts at the current point and ends at the current point plus ⟨*bend vector*⟩. At his point, it has its bend. The second half parabola starts at that bend point and end at point that is given by the bend plus ⟨*end vector*⟩.

If you set ⟨*end vector*⟩ to the null vector, you append only a half parabola that goes from the current point to the bend; by setting ⟨*bend vecotr*⟩ to the null vector, you append only a half parabola that goes to current point plus ⟨*end vector*⟩ and has its bend at the current point.

It is not possible to use this command to draw a part of a parabola that does not contain the bend.



```
\begin{pgfpicture}
  % Half-parabola going ``up and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpointorigin}{\pgfpoint{2cm}{4cm}}
  \color{red}
  \pgfusepath{stroke}

  % Half-parabola going ``down and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpoint{-2cm}{4cm}}{\pgfpointorigin}
  \color{blue}
  \pgfusepath{stroke}

  % Full parabola
  \pgfpathmoveto{\pgfpoint{-2cm}{2cm}}
  \pgfpathparabola{\pgfpoint{1cm}{-1cm}}{\pgfpoint{2cm}{4cm}}
  \color{orange}
  \pgfusepath{stroke}
\end{pgfpicture}
```

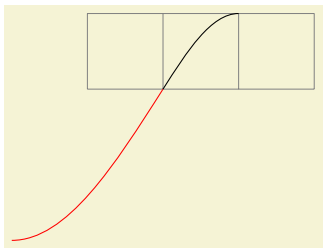
The command will apply coordinate transformations and update the bounding boxes.

17.10 Sine and Cosine Path Operations

Sine a cosine curves often need to be drawn and the following commands may help with this. However, they only allow you to append sine and cosine curves in intervals that are multiples of $\pi/2$.

\pgfpathsine{⟨*vector*⟩}

This command appends a sine curve in the interval $[0, \pi/2]$ to the current path. The sine curve is squeezed or strechted such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩.

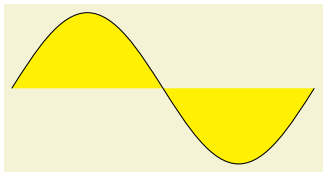


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,1);
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}

  \color{red}
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{-2cm}{-2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

\pgfpathcosine{⟨*vector*⟩}

This command appends a cosine curve in the interval $[0, \pi/2]$ to the current path. The curve is squeezed or strechted such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩. Using several sine and cosine operations in sequence allows you to produce a complete sine or cosine curve



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{-1cm}}
  \pgfpathsine{\pgfpoint{1cm}{-1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{1cm}}
  \pgfsetfillcolor{yellow}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

17.11 Plot Path Operations

There exist several commands for appending plots to a path. These commands are available through the package `pgfbaseplot`. They are documented in Section 25.

17.12 Rounded Corners

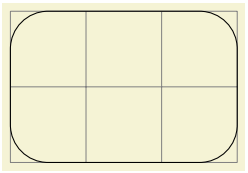
Normally, when you connect two straight line segments or when you connect two curves that end and start "at different angles" you get "sharp corners" between the lines or curves. In some cases it would be desirable that we get "rounded corners" instead. Thus, the lines or curves should be shortened a bit and then connected by arcs.

PGF offers an easy way to achieve this effect, by calling the following two commands.

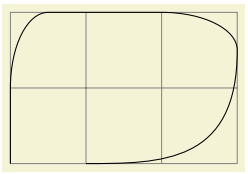
`\pgfsetcornersarced{`⟨*point*⟩`}`

This command causes all subsequent corners to be replaced by little arcs. The effect of this command lasts till the end of the current TEX scope.

The ⟨*point*⟩ dictates how large the corner arc will be. Consider a corner made by two lines l and r and assume that the line l comes first on the path. The x-dimension of the ⟨*point*⟩ decides by how much of the line l will be shortened, the y-dimension of ⟨*point*⟩ decides by how much the line r will be shortened. Then, the shortened lines are connected by an arc.



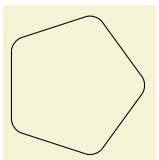
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfsetcornersarced{\pgfpoint{5mm}{5mm}}
  \pgfpathrectanglecorners{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```



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

  \pgfsetcornersarced{\pgfpoint{10mm}{5mm}}
  % 10mm entering,
  % 5mm leaving.
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{2cm}}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfpathcurveto
    {\pgfpoint{3cm}{0cm}}
    {\pgfpoint{2cm}{0cm}}
    {\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

If the x- and y-coordinates of ⟨*point*⟩ are the same and the corner is a right angle, you will get a perfect quarter circle (well, not quite perfect, but perfect up to six decimals). When the angle is not 90°, you only get a fair approximation.

More or less "all" corners will be rounded, even the corner generated by a `\pgfpathclose` command. (The author is a bit proud of this feature.)



```
\begin{pgfpicture}
  \pgfsetcornersarced{\pgfpoint{4pt}{4pt}}
  \pgfpathmoveto{\pgfpointpolar{0}{1cm}}
  \pgfpathlineto{\pgfpointpolar{72}{1cm}}
  \pgfpathlineto{\pgfpointpolar{144}{1cm}}
  \pgfpathlineto{\pgfpointpolar{216}{1cm}}
  \pgfpathlineto{\pgfpointpolar{288}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

To return to normal (unrounded) corners, use `\pgfsetcornersarced{\pgfpointorigin}`.

Note that the rounding will produce strange and undesirable effects if the lines at the corners are too short. In this case, the shortening may cause the lines to "suddenly extend over the other end" which is rarely desirable.

17.13 Internal Tracking of Bounding Boxes for Paths and Pictures

The path construction commands keep track of two bounding boxes: One for the current path, which is reset whenever the path is used and thereby flushed, and a bounding box for the current `{pgfpicture}`.

The bounding boxes are not accessible by "normal" macros. Rather, two sets of four dimension variables are used for this, all of which contain the letter `@`.

`\pgf@pathminx`

The minimum x-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxx`

The maxmimum x-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

`\pgf@pathminy`

The minimum y-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxy`

The maxmimum y-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

`\pgf@picminx`

The minimum x-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

`\pgf@picmaxx`

The maxmimum x-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

`\pgf@picminy`

The minimum y-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

`\pgf@picmaxy`

The maxmimum y-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

Each time a path construction command is called, the above variables are (globally) updated. To facilitate this, you can use the following command:

`\pgf@protocolsizes{`⟨*x-dimension*⟩`}{`⟨*y-dimension*⟩`}`

Updates all of the above dimension in such a way that the point specified by the two arguments is inside both bounding boxes. For the picture's bounding box this updating occurs only if `\ifpgf@relevantforpicturesize` is true, see below.

For the bounding box of the picture it is not always desirable that every path construction command affects this bounding box. For example, if you have just used a clip command, we do not want anything outside the clipping area to affect the boundinb box. For this reason, there exists a special "TEX if" that (locally) decides whether updating should be applied to the picture's bounding box. Clipping will set this if to false, as will certain other commands.

`\pgf@relevantforpicturesizefalse`

Suppresses updating of the picture's bounding box.

`\pgf@relevantforpicturesizetrue`

Causes updating of the picture's bounding box.

18 Using Paths

18.1 Overview

Once a path has been constructed, it can be *used* in different ways. For example, you can draw the path or fill it or use it for clipping.

Numerous graph parameters influence how a path will be rendered. For example, when you draw a path the line width is important as well as the dashing pattern. The options that govern how paths are drawn all start with `\pgfset`. *All options that influence how a path is rendered always influence the complete path.* Thus, it is not possible to draw part of a path using, say, a red color and drawing another part using a green color. To achieve such an effect, you must use two paths.

In detail, paths can be used in the following ways:

1. You can *stroke* (also known as *draw*) a path.
2. You can *fill* a path with a uniform color.
3. You can *clip* subsequent renderings against the path.
4. You can *shade* a path.
5. You can *use the path as bounding box* for the whole picture.

You can also perform any combination of the above, though it makes no sense to fill and shade a path at the same time.

To perform (a combination of) the first three actions, you can use the following command:

`\pgfusepath{`⟨*actions*⟩`}`

Applies the given ⟨*actions*⟩ to the current path. Afterwards, the current path is (globally) empty. The following actions are possible:

- `fill` fills the path. See Section 18.3 for further details.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `stroke` strokes the path. See Section 18.2 for further details.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

- `clip` clips all subsequent drawings against the path. See Section 18.4 for further details.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke,clip}
  \pgfpathcircle{\pgfpoint{1cm}{1cm}}{0.5cm}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `discard` discards the path, that is, it is not used at all. Giving this option (alone) has the same effect as giving an empty options list.

When more than one of the first three actions are given, they are applied in the above ordering, regardless of their ordering in ⟨*actions*⟩. Thus, `{stroke,fill}` and `{fill,stroke}` have the same effect.

To shade a path, use the `\pgfshadepath` command, which is explained in Section 23.

18.2 Stroking a Path

When you use `\pgfusepath{stroke}` to stroke a path, several graphic parameters influence thow the path is drawn. The commands for setting these parameters are explained in the following.

Note that all graphic parameters apply to the path as a whole, never only to a part of it.

All graphic parameters are local to the current `{pgfscope}`, but they persists past TEX groups, *except* for the iterior rule (even-odd or non-zero) and the arrow type. The latter to graphic parameter only persist till the end of the current TEX group, but this may change in the future, so do not count on this.

18.2.1 Graphic Parameter: Line Width

`\pgfsetlinewidth{`⟨*line width*⟩`}`

This command sets the line width for subsequent strokings (in the current `pgfscope`). The line width is given as a normal TEX dimension like `0.4pt` or `1mm`.

```
\begin{pgfpicture}
  \pgfsetlinewidth{1mm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{2\pgflinewidth} % double in size
  \pgfpathmoveto{\pgfpoint{0mm}{5mm}}
  \pgfpathlineto{\pgfpoint{2cm}{5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgflinewidth`

You can access the current line width via the TEX dimension `\pgflinewidth`. It will be set to the correct line width, that is, even when a TEX group closed, the value will be correct since it is set globally, but when a `{pgfscope}` closes, the value is set to the correct value it had before the scope.

18.2.2 Graphic Parameter: Caps and Joins

`\pgfsetbuttcap`

Sets the line cap to a butt cap. See Section 8.2.1 for an explanation of what this is.

`\pgfsetroundcap`

Sets the line cap to a round cap. See again Section 8.2.1.

`\pgfsetrectcap`

Sets the line cap to a square cap. See again Section 8.2.1.

`\pgfsetroundjoin`

Sets the line join to a round join. See again Section 8.2.1.

`\pgfsetbeveljoin`

Sets the line join to a bevel join. See again Section 8.2.1.

`\pgfsetmiterjoin`

Sets the line join to a miter join. See again Section 8.2.1.

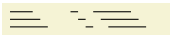
`\pgfsetmiterlimit{`⟨*miter limit factor*⟩`}`

Sets the miter limit to ⟨*miter limit factor*⟩. See again Section 8.2.1.

18.2.3 Graphic Parameter: Dashing

\pgfsetdash{⟨*list of even length of dimensions*⟩}{⟨*phase*⟩}

Sets the dashing of a line. The first entry in the list specifies the length of the first solid part of the list. The second entry specifies the length of the following gap. Then comes the length of the second solid part, following by the length of the second gap, and so on. The ⟨*phase*⟩ specifies where the first solid part starts relative to the beginning of the line.



```
\begin{pgfpicture}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0cm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.1cm}
  \pgfpathmoveto{\pgfpoint{0mm}{1mm}}
  \pgfpathlineto{\pgfpoint{2cm}{1mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.2cm}
  \pgfpathmoveto{\pgfpoint{0mm}{2mm}}
  \pgfpathlineto{\pgfpoint{2cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

Use `\pgfsetdash{}{0pt}` to get a solid dashing.

18.2.4 Graphic Parameter: Stroke Color

\pgfsetstrokecolor{⟨*color*⟩}

Sets the color used for stroking lines to ⟨*color*⟩, where ⟨*color*⟩ is a LaTeX color like `red` or `black!20!red`. Unlike the `\color` command, the effect of this command last till the end of the current `{pgfscope}` and not till the end of the current TeX group.

The color used for stroking may be different from the color used for filling. However, a `\color` command will always "immediately override" any special settings for the stroke and fill colors.

In plain TeX, this command will also work, but the problem of *defining* a color arises. After all, plain TeX does not provide LaTeX colors. For this reasong, PGF implements a minimalistic "emulation" of the `\defincolor`, `\colorlet`, and `\color` commands. Only grayscale and rgb colors are supported. For most cases this turns out to be enough.

```
\begin{pgfpicture}
  \pgfsetlinewidth{1pt}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \pgfsetstrokecolor{black}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \color{red}
  \pgfpathcircle{\pgfpoint{2cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
\end{pgfpicture}
```

\pgfsetcolor{⟨*color*⟩}

Sets both the stroke and fill color. The difference to the normal `\color` command is that the effect lasts till the end of the current `{pgfscope}`, not only till the end of the current TeX group.

18.2.5 Graphic Parameter: Arrows

After a path has been drawn, PGF can add arrows at the ends. Currently, it will only add arrows correctly at the end of paths the consist of a single open part. For other paths, like closed paths or path consisting of multiple parts, the result is not defined.

\pgfsetarrowsstart{⟨*arrow kind*⟩}

Sets the arrow king used at the start of a (possibly curved) path. When this option is used, the line will often be slightly shortened to ensure that the tip of the arrow will exactly "touch" the "real" start of the line.

To “clear” the start arrow, say `\pgfsetstartarrow{}`.

```
\begin{pgfpicture}
  \pgfsetarrowsstart{latex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
  \pgfsetarrowsstart{to}
  \pgfpathmoveto{\pgfpoint{0cm}{2mm}}
  \pgfpathlineto{\pgfpoint{1cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The effect of this command persists only till the end of the current TEX scope.

The different possible arrow kinds are explained in Section 20.

`\pgfsetarrowsend{`⟨*arrow kind*⟩`}`

Sets the arrow king used at the end of a path.


```
\begin{pgfpicture}
  \pgfsetarrowsstart{latex}
  \pgfsetarrowsend{to}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetarrows{`⟨*start kind*⟩`-`⟨*end kind*⟩`}`

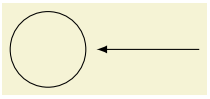
Sets the start arrow kind to ⟨*start kind*⟩ and the end kind to ⟨*end kind*⟩.


```
\begin{pgfpicture}
  \pgfsetarrows{latex-to}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetshortenstart{`⟨*dimension*⟩`}`

This command will shortend the start of every stroked path by the given dimension. This shortening is done in addition to automatic shortening done by a start arrow, but it can be used even if no start arrow is given.

This command is useful if you wish arrows or line to “stop shortly before” a given point.



```
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{5mm}
  \pgfusepath{stroke}
  \pgfsetarrows{latex-}
  \pgfsetshortenstart{4pt}
  \pgfpathmoveto{\pgfpoint{5mm}{0cm}} % would be on the circle
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetshortenend{`⟨*dimension*⟩`}`

Works like `\pgfsetshortenstart`.

18.3 Filling a Path

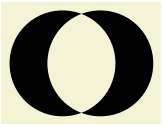
Filling a path means coloring every interior point of the path with the current fill color. It is not always obivous whether a point is “inside” a path when the path is self-intersecting and/or consists or multiple parts. In this case either the nonzero winding number rule or the even-odd crossing number rule is used to decide, which points lie “inside.” These rules are explained in Section 8.3.

18.3.1 Graphic Parameter: Interior Rule

You can set which rule is used using the following commands:

\pgfseteorule

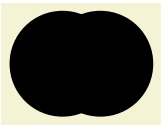
Dictates, that the even-odd rule is used in subsequent fillings in the current *TEX scope*. Thus, for once, the effect of this command does not persist past the current TEX scope.



```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

\pgfsetnonzerorule

Dictates, that the nonzero winding number rule is used in subsequent fillings in the current TEX scope. This is the default.



```
\begin{pgfpicture}
  \pgfsetnonzerorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

18.3.2 Graphic Parameter: Filling Color

\pgfsetfillcolor{⟨*color*⟩}

Sets the color used for filling paths to ⟨*color*⟩. Like the stroke color, the effect lasts only till the next use of `\color`.

18.4 Clipping a Path

When you add the `clip` option, the current path is used for clipping subsequent drawings. The same rule as for filling is used to decide whether a point is inside or outside the path, that is, either the even-odd rule or the non-zero rule.

Clipping never enlarges the clipping area. Thus, when you clip against a certain path and then clip again against another path, you clip against the intersection of both.

The only way to enlarge the clipping path is to end the `{pgfscope}` in which the clipping was done. At the end of a `{pgfscope}` the clipping path that was in force at the beginning of the scope is reinstalled.

18.5 Using a Path as Bounding Box

When you add the `use as bounding box` option, the bouding box of the picture will be enlarged such that the path in encompassed, but any *subsequent* paths of the current TEX scope will not have any effect in the size of the bounding box. Typically, you use this command at the very beginning of a `{pgfpicture}` environment.


```
Left
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{use as bounding box} % draws nothing

  \pgfpathcircle{\pgfpointorigin}{2ex}
  \pgfusepath{stroke}
\end{pgfpicture}
right.
```

19 Hierarchical Structures: Package, Environments, Scopes, and Text

19.1 Overview

PGF uses two kinds of hierarchical structuring: First, the package itself is structured hierarchically, consisting of different packages that build on top of each other. Second, PGF allows you to structure your graphics hierarchically using environments and scopes.

19.1.1 The Hierarchical Structure of the Package

The PGF system consists of several layers:

System layer. The lowest layer is called the *system layer*, though it might also be called "driver layer" or perhaps "backend layer." Its job is to provide an abstraction of the details of which driver is used to transform the `.dvi` file. The system layer is implemented by the package `pgfsys`, which will load appropriate driver files as needed.

The system layer is documented in Part V.

Basic layer. The basic layer is loaded by the package `pgf`. Some applications do not need all of the functionality of the basic layer, so it is possible to load only the `pgfcore` and some other packages starting with `pgfbase`.

The basic layer is documented in the present part.

Frontend layer. The frontend layer is not loaded by a single packages. Rather, different packages, like Ti*k*Z or PGFPICT2E, are different frontends to the basic layer.

The Ti*k*Z frontend is documented in Part II.

Each layer will automatically the necessary files of the layers below it.

In addition to the packages of these layers, there are also some library packages. These packages provide additional definitions of things like new arrow tips or new plot handlers.

The library packages are documented in Part III.

19.1.2 The Hierarchical Structure of Graphics

Graphics in PGF are typically structured hierarchically. Hierarchical structuring can be used to identify groups of graphical elements that are to be treated "in the same way." For example, you might group together a number of paths, all of which are to be drawn in red. Then, when you decide later on that you like them to be drawn in, say, blue, all you have to do is to change the color once.

The general mechanism underlying hierarchical structuring is known as *scoping* in computer science. The idea is that all changes to the general "state" of the graphic that are done inside a scope are local to that scope. So, if you change the color inside a scope, this does not affect the color used outside the scope. Likewise, when you change the line width in a scope, the line width outside is not changed, and so on.

There are different ways of starting and ending scopes of graphic parameters. Unfortunately, these scopes are sometimes "in conflict" with each other and it is sometimes not immediately clear which scopes apply. In essence, the following scoping mechanisms are available:

1. The "outermost" scope supported by PGF is the `{pgfpicture}` environment. All changes to the graphic state done inside a `{pgfpicture}` are local to that picture.

   In general, it is *not* possible to set graphic parameters globally outside any `{pgfpicture}` environments. Thus, you can *not* say `\pgfsetlinewidth{1pt}` at the beginning of your document to have a default line width of one point. Rather, you have to (re)set all graphic parameters inside each `{pgfpicture}`. (If this is too bothersome, try defining some macro that does the job for you.)

2. Inside a `{pgfpicture}` you can use a `{pgfscope}` environment to keep changes of the graphic state local to that environment.

   The effect of commands that change the graphic state are local to the current `{pgfscope}` but not always to the current TEX group. Thus, if you open a TEX group (some text in curly braces) inside a

`{pgfscope}`, and if you change, for example, the dash pattern, the effect of this changed dash pattern will persist till the end of the `{pgfscope}`.

Unfortunately, this is not always the case. *Some* graphic parameters only persist till the end of the current TEX group. For example, when you use `\pgfsetarrows` to set the arrow tip kind inside a TEX group, the effect lasts only till the end of the current TEX group.

3. Some graphic parameters are not scoped by `{pgfscope}` but "already" by TEX groups. For example, the effect of coordinate transformation commands is always local to the current TEX group.

   Since every `{pgfscope}` automatically creates a TEX group, all graphic parameters that are local to the current TEX group are also local to the current `{pgfscope}`.

4. Some graphic parameters can only be scoped using TEX groups, since in some situations it is not possible to introduce a `{pgfscope}`. For example, a path always has to be completely constructed and used in the same `{pgfscope}`. However, we might wish to have different coordinate transformations apply to different points on the path. In this case, we can use TEX groups to keep the effect local, but we could not use `{pgfscope}`.

5. The `\pgftext` command can be used to create a scope in which TEX "escapes back" to normal TEX mode. The text passed to the `\pgftext` is "heavily guarded" against having any effect on the scope in which it is used. For example, it is possibly to use another `{pgfpicture}` environment inside the argument of `\pgftext`.

Most of the complications can be avoided if you stick to the following rules:

- Give graphic commands only inside `{pgfpicture}` environments.
- Use `{pgfscope}` to structure graphics.
- Do not use TEX groups inside graphics, *except* for keeping the effect of coordinate transformations local.

19.2 The Hierarchical Structure of the Package

Before we come to the structuring commands provided by PGF to structure your graphics, let us first have a look at the structure of the package itself.

19.2.1 The Main Package

To use PGF, include the following package:

```
\usepackage{pgf} % LaTeX
\input pgf.tex   % plain TeX
\input pgf.tex   % ConTeX
```

This package loads the complete "basic layer" of PGF. That is, it will load all of the commands described in the current part of this manual, but it will not load frontends like Ti*k*Z.

In detail, this package will load the following packages, each of which can also be loaded individually:

- `pgfsys`, which is the lowest layer of PGF and which is always needed. This file will read `pgf.cfg` to discern which driver is to be used. See Section 27.1 for details.
- `pgfcore`, which is the central core of PGF and which is always needed unless you intend to write a new basic layer from scratch.
- `pgfbaseimage`, which provides commands for declaring and using images, An example is `\pgfuseimage`.
- `pgfbaseshapes`, which provides commands for declaring and using shapes. An example is `\pgfshape`.
- `pgfbaseplot`, which provides commands for plotting functions.

Including any of the last three packages will automatically load the first two.

In LaTeX, the package takes two options:

`\usepackage[draft]{pgf}`

When this option is set, all images will be replaced by empty rectangles. This can speedup compilation.

`\usepackage[strict]{pgf}`

This option will suppress loading of a large number of compatibility commands.

19.2.2 The Core Package

```
\usepackage{pgfcore} % LaTeX
\input pgfcore.tex   % plain TeX
\input pgfcore.tex   % ConTeX
```

This package defines all of the basic layer's commands, except for the commands defined in the additional packages like `pgfbaseplot`. Typically commands defined by the core include `\pgfusepath` or `\pgfpoint`. The core is internally structured into several subpackages, but the subpackages cannot be loaded individually since they are all "interrelated."

19.2.3 The Optional Basic Layer Packages

The `pgf` package automatically loads the following packages, but you can also load them individually (all of them automatically include the core):

- `pgfbaseshapes` This package provides commands for drawing nodes and shapes. These commands are explained in Section 21.
- `pgfbaseplot` This package provides commands for plotting function. The commands are explained in Section 25.
- `pgfbaseimage` This package provides commands for including (external) images. The commands are explained in Section 24.

19.3 The Hierarchical Structure of the Graphics

19.3.1 The Main Environment

Most, but not all, commands of the PGF package must be given within a `{pgfpicture}` environment. The only commands that (must) be given outside are commands having to do with including images (like `\pgfuseimage`) and with inserting complete shadings (like `\pgfuseshading`). However, just to keep life entertaining, the `\pgfshadepath` command must be given *inside* a `{pgfpicture}` environment.

`\begin{pgfpicture}`
⟨*environment contents*⟩
`\end{pgfpicture}`

This environment will insert a TeX box containing the graphic drawn by the ⟨*environment contents*⟩ at the current position. Note that Ti*k*Z redefines this environment so that it takes an optional options argument.

The size of the bounding box. The size of the box is determined in the following manner: While PGF parses the ⟨*environment contents*⟩, it keeps track of a bounding box for the graphic. Essentially, this bounding box is the smallest box that contains all coordinates mentioned in the graphics. Some coordinates may be "mentioned" by PGF itself; for example, when you add circle to the current path, the support points of the curve making up the circle are also "mentioned" despite the fact that you will not "see" them in your code.

Once the ⟨*environment contents*⟩ has been parsed completely, a TeX box is created whose size is the size of the computed bounding box and this box is inserted at the current position.

Hello ▭ World!

```
Hello \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} World!
```

Sometimes, you may need more fine-grained control over the size of the bounding box. For example, the computed bounding box may be too large or you intensionally wish the box to be "too small." In these cases, you can use the command `\pgfusepath{use as bounding box}`, as described in Section 18.5.

The baseline of the bounding box. When the box containing the graphic is inserted into the normal text, the baseline of the graphic is normally at the bottom of the graphic. For this reason, the following two sets of code lines have the same effect, despite the fact that the second graphic uses "higher" coordinates than the first:

Rectangles and .

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture}.
```

You can change the baseline using the `\pgfsetbaseline` command, see below.

Rectangles and .

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture}.
```

Including text and images in a picture. You cannot directly include text and images in a picture. Thus, you should *not* simply write some text in a `{pgfpicture}` or use a command like `\includegraphics` or even `\pgfimage`. In all these cases, you need to place the text inside a `\pgftext` command. This will "escape back" to normal TEX mode, see Section 19.3.3 for details.

`\pgfpicture`
⟨*environment contents*⟩
`\endpgfpicture`

The plain TEX version of the environment. Note that in this version, also, a TEX group is created around the environment.

`\pgfsetbaseline{`⟨*dimension*⟩`}`

This command specifies a y-coordinate of the picture that should be used as the baseline of the whole picture. When a PGF picture has been typeset completely, PGF must decide at which height the baseline of the picture should lie. Normally, the baseline is set to the y-coordinate of the bottom of the picture, but it is often desirable to use another height.

Text ,  ,  ,  .

```
Text \tikz{\pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{0pt}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{.5ex}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{-1ex}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}}.
```

19.3.2 Graphic Scope Environments

Inside a `{pgfpicture}` environment you can substructure your picture using the following environment:

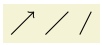
`\begin{pgfscope}`
⟨*environment contents*⟩
`\end{pgfscope}`

All changes to the graphic state done inside this environment are local to the environment. The graphic state includes the following:

- The line width.
- The stroke and fill colors.
- The dash pattern.
- The line join and cap.
- The miter limit.
- The canvas transformation matrix.
- The clipping path.

Other parameters may also influence how graphics are rendered, but they are *not* part of the graphic state. For example, the arrow tip kind is not part of the graphic state and the effect of commands setting the arrow tip kind are local to the current TEX group, not to the current `{pgfscope}`. However, since `{pgfscope}` starts and ends a TEX group automatically, a `{pgfscope}` can be used to limit the effect of, say, commands that set the arrow tip kind.

```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetlinewidth{2pt}
      \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathrectangle{\pgfpoint{3ex}{0ex}}{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathrectangle{\pgfpoint{6ex}{0ex}}{\pgfpoint{2ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetarrows{-to}
      \pgfpathmoveto{\pgfpointorigin}\pgfpathlineto{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathmoveto{\pgfpoint{3ex}{0ex}}\pgfpathlineto{\pgfpoint{5ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathmoveto{\pgfpoint{6ex}{0ex}}\pgfpathlineto{\pgfpoint{7ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

At the start of the scope, the current path must be empty, that is, you cannot open a scope while constructing a path.

It is usually a good idea *not* to introduce TEX groups inside a `{pgfscope}` environment.

`\pgfscope`
⟨*environment contents*⟩
`\endpgfscope`

Plain TEX version of the `{pgfscope}` environment.

The following scopes also encapsulate certain properties of the graphic state. However, they are typically not used directly by the user.

`\begin{pgfinterruptpath}`
⟨*environment contents*⟩
`\end{pgfinterruptpath}`

This environment can be used to temporarily interrupt the construction of the current path. The effect will be that the path currently under construction will be "stored away" and restored at the end of the environment. Inside the environment you can construct a new path and do something with it.

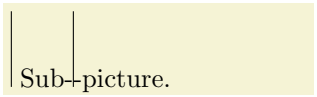
An example application of this environment is the arrow tip caching. Suppose you ask PGF to use a specific arrow tip kind. When the arrow tip needs to be rendered for the first time, PGF will "cache"

the path that makes up the arrow tip. To do so, it interrupts the current path construction and then protocols the path of the arrow tip. The {pgfinterruptpath} environment is used to ensure that this does not interfere with the path to which the arrow tips should be attached.

This command does *not* install a {pgfscope}. In particular, it does not call any \pgfsys@ commands at all, which would, indeed, be dangerous in the middle of a path construction.

\begin{pgfinterruptpicture}
⟨*environment contents*⟩
\end{pgfinterruptpicture}

This environment can be used to temporarily interrupt a {pgfpicture}. However, the environment is intended only to be used at the beginning and end of a box that is (later) inserted into a {pgfpicture} using \pgfqbox. You cannot use this environment directly inside a {pgfpicture}.




```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}} % In the middle of path, now
  \newbox\mybox
  \setbox\mybox=\hbox{
    \begin{pgfinterruptpicture}
      Sub-\begin{pgfpicture} % a subpicture
        \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
        \pgfpathlineto{\pgfpoint{1cm}{1cm}}
        \pgfusepath{stroke}
      \end{pgfpicture}-picture.
    \end{pgfinterruptpicture}
  }
  \ht\mybox=0pt
  \wd\mybox=0pt
  \dp\mybox=0pt
  \pgfqbox{\box\mybox}%
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}\hskip3.9cm
```

19.3.3 Inserting Text and Images

Often, you may wish to add normal TEX text at a certain point inside a {pgfpicture}. You cannot do so "directly," that is, you cannot simply write this text inside the {pgfpicture} environment. Rather, you must pass the text as an argument to the \pgftext command.

You must *also* use the \pgftext command to insert an image or a shading into a {pgfpicture}.

\pgftext[⟨*options*⟩]{⟨*text*⟩}

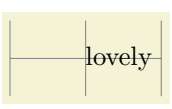
This command will typeset ⟨*text*⟩ in normal TEX mode and insert the resulting box into the {pgfpicture}. The bounding box of the graphic will be updated so that all of the text box is inside. Be default, the text box is centered at the origin, but this can be changed either by giving appropriate ⟨*options*⟩ or by applying an appropriate coordinate transformation beforehand.

The ⟨*text*⟩ may contain verbatim text. (In other words, the ⟨*text*⟩ "argument" is not a normal argument, but is put in a box and some \aftergroup hackery is used to find the end of the box.)

PGF's current (high-level) coordinate transformation is synchronized with the canvas transformation matrix temporarily when the text box is inserted. The effect is that if there is currently a high-level rotation of, say, 30 degrees, the ⟨*text*⟩ will also be rotated by thirty degrees. If you do not want this effect, you have to (possibly temporarily) reset the high-level transformation matrix.

The following ⟨*options*⟩ may be given as conveniences:

- left causes the text box to be placed such that its left border is on the origin.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[left] {lovely}}
```

- `right` causes the text box to be placed such that its right border is on the origin.


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[right] {lovely}}
```

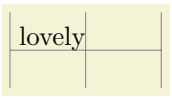
- `top` causes the text box to be placed such that its top is on the origin. This option can be used together with the `left` or `right` option.


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[top] {lovely}}
```


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[top,right] {lovely}}
```

- `bottom` causes the text box to be placed such that its bottom is on the origin.

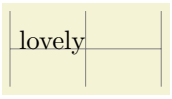

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[bottom] {lovely}}
```




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[bottom,right] {lovely}}
```

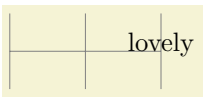
- `base` causes the text box to be placed such that its baseline is on the origin.


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base] {lovely}}
```



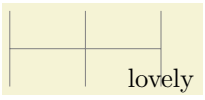

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,right] {lovely}}
```

- `at`=⟨*point*⟩ Translates the origin (that is, the point where the text is shown) to ⟨*point*⟩.



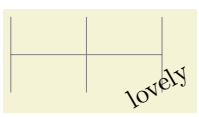

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,at={\pgfpoint{1cm}{0cm}}] {lovely}}
```

- `x`=⟨*dimension*⟩ Translates the origin by ⟨*dimension*⟩ along the x-axis.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,x=1cm,y=-0.5cm] {lovely}}
```

- `y`=⟨*dimension*⟩ works like the `x` option.
- `rotate`=⟨*degree*⟩ Rotates the coordinate system by ⟨*degree*⟩. This will also rotate the text box.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,x=1cm,y=-0.5cm,rotate=30] {lovely}}
```

20 Arrow Tips

20.1 Overview

20.1.1 When Does PGF Draw Arrows?

PGF offers an interface for placing *arrow tips* at the end of lines. The interface works as follows:

1. You assign a name to a certain kind of arrow tips. For example, the arrow tip `latex` is the arrow tip used by the standard LaTeX picture environment; the arrow tip `to` looks like the tip of the arrow in TeX's `\to` command; and so on.

   This is done once at the beginning of the document.

2. Inside some picture, at some point you specify that in the current scope from now on you would like tips of, say, kind `to` to be added at the end and/or beginning of all paths.

   When an arrow kind has been installed and when PGF is about to stroke a path, the following things happen:

   (a) The beginning and/or end of the path is shortened appropriately.
   (b) The path is stroked.
   (c) The arrow tip is drawn at the beginning and/or end of the path, appropriately rotated and appropriately resized.

In the above description, there are a number of "appropriatelies." The exact details are not quite trivial and described later on.

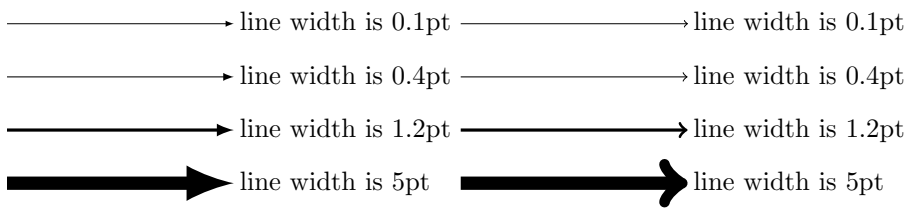
20.1.2 Meta-Arrows

In PGF, arrows are "meta-arrows" in the same way that fonts in TeX are "meta-fonts." When a meta-arrow is resized, it is not simply scaled, but a possibly complicated transformation is applied to the size.

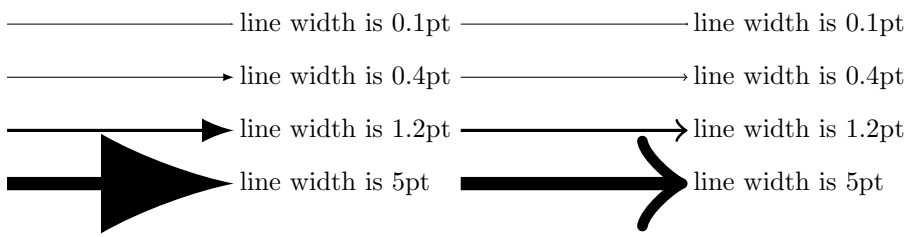
A meta-font is not one particular font at a specific size with a specific stroke width (and with a large number of other parameters being fixed). Rather, it is a "blueprint" (actually, more like a program) for generating such a font at a particular size and width. This allows the designer of a meta-font to make sure that, say, the font is somewhat thicker and wider at very small sizes. To appreciate the difference: Compare the following texts: "Berlin" and "Berlin". The first is a "normal" text, the second is the tiny version scaled by a factor of two. Obviously, the first look better. Now, compare "Berlin" and "Berlin". This time, the normal text was scaled down, while the second text is a "normal" tiny text. The second text is easier to read.

PGF's meta-arrows work in a similar fashion: The shape of an arrow tip can vary according to the line width of the arrow tip is used. Thus, an arrow tip drawn at a line width of 5pt will typically *not* be five times as large as an arrow tip of line width 1pt. Instead, the size of the arrow will get bigger only slowly as the line width increases.

To appreciate the difference, here are the `latex` and `to` arrows, as drawn by PGF at four different sizes:



Here, by comparison, is the same arrow when it is simply "resized" (as done by most programs):



As can be seen, simple scaling produces arrow tips that are way too large at larger sizes and way too small at smaller sizes.

20.2 Declaring an Arrow Tip Kind from Scratch

To declare an arrow kind "from scratch," the following command is used:

\pgfarrowsdeclare{⟨*start name*⟩}{⟨*end name*⟩}{⟨*extend code*⟩}{⟨*arrow tip code*⟩}

This command declares a new arrow kind. An arrow kind has two names, which will typically be the same. When the arrow tip needs to be drawn, the ⟨*arrow tip code*⟩ will be invoked, but the canvas trasformation is setup beforehand to a rotation such that when an arrow tip pointing right is specified, the arrow tip that is actually drawn points in the direction of the line.

Naming the arrow kind. The ⟨*start name*⟩ is the name used for the arrow when it is at the start of a path, the ⟨*end name*⟩ is the name used at the end of a path. For example, the arrow kind that looks like a paranthesis has the ⟨*start name*⟩ `(` and the ⟨*end name*⟩ `)` so that you can say `\pgfsetarrows{(-)}` to specify that you want paranthesis arrows and both ends.

The ⟨*end name*⟩ and ⟨*start name*⟩ can be quite arbitrary and may contain spaces.

Basics of the arrow tip code. Let us next have a look at the ⟨*arrow tip code*⟩. This code will be used to draw the arrow when PGF thinks this is necessary. The code should draw an arrow that "points right," which means that is should draw an arrow at the end of a line coming from the left and ending at the origin.

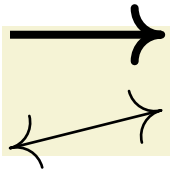
As an example, suppose we wanted to declare an arrow tip consisting of two arcs, that is, we want the arrow tip to look more or less like the red part of the following picture:

```
\begin{tikzpicture}[line width=3pt]
  \draw (-2,0) -- (0,0);
  \draw[red,join=round,cap=round]
        (-10pt,10pt) arc (180:270:10pt) arc (90:180:10pt);
\end{tikzpicture}
```

We could use the following as ⟨*arrow tip code*⟩ for this:

```
\pgfarrowsdeclare{arcs}{arcs}{...}
{
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-10pt}{10pt}}
  \pgfpatharc{180}{270}{10pt}
  \pgfpatharc{90}{180}{10pt}
  \pgfusepathqstroke
}
```

Indeed, when the `...` is set appropriately (in a moment), we can write the following:



```
\begin{tikzpicture}
  \draw[-arcs,line width=3pt] (-2,0)  -- (0,0);
  \draw[arcs-arcs,line width=1pt] (-2,-1.5) -- (0,-1);
\end{tikzpicture}
```

As can be seen in the second example, the arrow tip is automatically rotated as needed when the arrow is drawn. This is achieved by a canvas rotation.

Special considertations about the arrow tip code. There are several things you need to be aware of when designing arrow tip code:

- Inside the code, you may not use the `\pgfusepath` command. The reason is that this command internally calls arrow construction commands, which something you obviously do not want to happen.

  Instead of `\pgfusepath`, use the quick versions. Typically, you will use `\pgfusepathqstroke`, `\pgfusepathqfill`, or `\pgfusepathqfillstroke`.
- The code will be executed only once, namely the first time the arrow tip needs to be drawn. The resulting low-level driver commands are protocolled and stored away. In all subsequent uses of the arrow tip, the protocolled code is directly inserted.

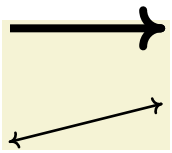
- However, the code will be executed anew for each line width. Thus, an arrow of line width 2pt may result in a different protocol than the same arrow for a line width of 0.4pt.
- If you stroke the path that you construct, you should first set the dashing to solid and setup fixed joins and caps, as needed. This will ensure that the arrow tip will always look the same.
- When the arrow tip code is executed, it is automatically put inside a low-level scope, so nothing will "leak out" from the scope.
- The high-level coordinate transformation matrix will be set to the identity matrix when the code is executed for the first time.

Designing meta-arrows. The ⟨*arrow tip code*⟩ should adjust the size of the arrow in accordance with the line width. For a small line width, the arrow tip should be small, for a large line width, it should be larger. However, the size of the arrow typically *should not* grow linearly with the line width. On the other hand, the size of the arrow head typically *should* grow "a bit" with the line width.

For these reasons, PGF will not simply executed your arrow code within a scaled scope, where the scaling depends on the line width. Instead, you ⟨*arrow tip code*⟩ is reexecuted again for each different line width.

In our example, we could use the following code for the new arrow tip kind `arc'` (note the prime):

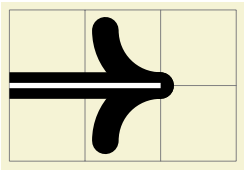
```
\newdimen\arrowsize
\pgfarrowsdeclare{arcs'}{arcs'}{...}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
  \pgfpatharc{180}{270}{4\arrowsize}
  \pgfpatharc{90}{180}{4\arrowsize}
  \pgfusepathqstroke
}
```



```
\begin{tikzpicture}
  \draw[-arcs',line width=3pt] (-2,0)  -- (0,0);
  \draw[arcs'-arcs',line width=1pt] (-2,-1.5) -- (0,-1);
\end{tikzpicture}
```

However, sometimes, it can also be useful to have arrows that do not resize at all when the line width changes. This can be achieved by givin absolute size coordinates in the code, as done for `arc`. On the other hand, you can also have the arrow resize linearly with the line width by specifying all coordinates as multiples of `\pgflinewidth`.

The left and right extend. Let us have another look at the exact left and right "ends" of our arrow tip. Let us draw the arrow tip `arc'` at a very large size:



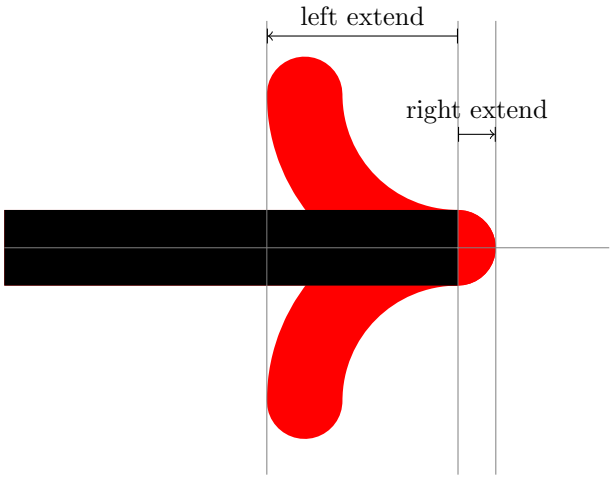
```
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
  \draw[line width=10pt,-arcs'] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
\end{tikzpicture}
```

As one can see, the arrow tip does not "touch" the origin as it should, but protrudes a little over the origin. One remedy to this undesirable effect is to change the code of the arrow tip such that everything is shifted half an `\arrowsize` to the left. While this will cause the arrow tip to touch the origin, the line itself will then interfere with the arrow: The arrow tip will be partly "hidden" by the line itself.

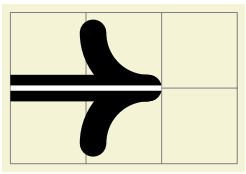
PGF uses a different approach to solving the problem: The ⟨*extend code*⟩ argument can be used to "tell" PGF how much the arrow protrudes over the origin. The argument is also used to tell PGF where the "left" end of the arrow is. However, this number is important only when the arrow is being reversed.

Once PGF knows the right extend of an arrow kind, it can *shorten* lines by this amount when drawing arrows.

Here is a picture that shows what the visualizes the extends. The arrow tip itself is shown in red once more:



The ⟨*extend code*⟩ is normal TEX code that is executed whenever PGF wants to know how far the arrow tip will protrude to the right and left. The code should call the following two commands: \pgfarrowsrightextend and \pgfarrowsleftextend. Both arguments take one argument that contain the size. Here is the final code for the `arc''` arrow tip:



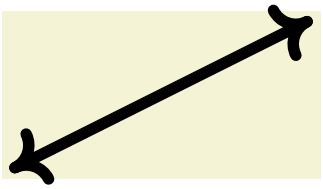
```
\pgfarrowsdeclare{arcs''}{arcs''}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfarrowsleftextend{-4\arrowsize-.5\pgflinewidth}
  \pgfarrowsrightextend{.5\pgflinewidth}
}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
  \pgfpatharc{180}{270}{4\arrowsize}
  \pgfusepathqstroke
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{90}{180}{4\arrowsize}
  \pgfusepathqstroke
}
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
  \draw[line width=10pt,-arcs''] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
\end{tikzpicture}
```

20.3 Declaring a Derived Arrow Tip Kind

It is possible to declare arrow kinds in terms of existing ones. For these command to work correctly, the left and right extends must be set correctly.

\pgfarrowsdeclarealias{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

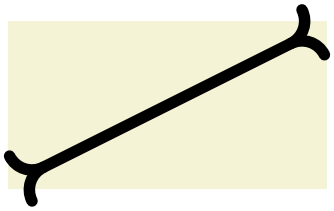
This command can be used to create an alias (another name) for an existing arrow kind.



```
\pgfarrowsdeclarealias{<}{>}{arcs''}{arcs''}%
\begin{pgfpicture}
  \pgfsetarrows{<->}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfarrowsdeclarereversed{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

This command creates a new arrow kind that is the "reverse" of an existing arrow kind. The (automatically cerated) code of the new arrow kind will contain a flip of the canvas and the meanings of the left and right extend will be reversed.

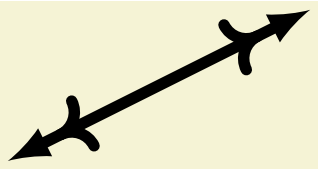


```
\pgfarrowsdeclarereversed{arcs reversed}{arcs reversed}{arcs''}{arcs''}%
\begin{pgfpicture}
  \pgfsetarrows{arcs reversed-arcs reversed}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfarrowsdeclarecombine*[⟨*offset*⟩]{⟨*start name*⟩}{⟨*end name*⟩}{⟨*first start name*⟩}{⟨*first end name*⟩}{⟨*second start name*⟩}{⟨*second end name*⟩}

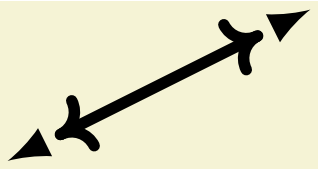
This command creates a new arrow kind that combines two existing arrow kinds. The first arrow kind is the "innermost" arrow kind, the second arrow kind is the "outermost."

The code for the combined arrow kind will install a canvas translation before the innermost arrow kind in drawn. This translation is calculated such that the right tip of the innermost arrow touches the right end of the outermost arrow. The optional ⟨*offset*⟩ can be used to increase (or decrease) the distance between the inner and outermost arrow.



```
\pgfarrowsdeclarecombine[\pgflinewidth]
  {combined}{combined}{arcs''}{arcs''}{latex}{latex}%
\begin{pgfpicture}
  \pgfsetarrows{combined-combined}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

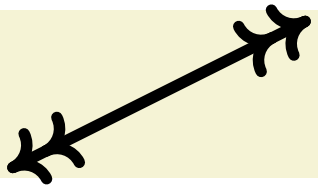
In the star variant, the end of the line is not in the outermost arrow, but inside the innermost arrow.



```
\pgfarrowsdeclarecombine*[\pgflinewidth]
  {combined'}{combined'}{arcs''}{arcs''}{latex}{latex}%
\begin{pgfpicture}
  \pgfsetarrows{combined'-combined'}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfarrowsdeclaredouble[⟨*offset*⟩]{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

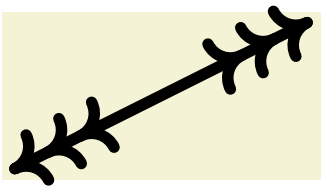
This command is a shortcut for combining an arrow kind with itself.



```
\pgfarrowsdeclaredouble{<<}{>>}{arcs''}{arcs''}%
\begin{pgfpicture}
  \pgfsetarrows{<<->>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfarrowsdeclaretriple[⟨*offset*⟩]{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

This command is a shortcut for combining an arrow kind with itself and then again.



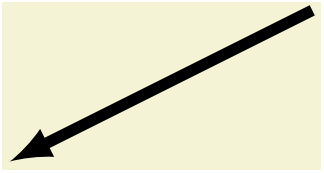
```
\pgfarrowsdeclaretriple{<<<}{>>>}{arcs''}{arcs''}%
\begin{pgfpicture}
  \pgfsetarrows{<<<->>>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

20.4 Using an Arrow Kind

The following commands install the arrow kind that will be used when stroking is done.

\pgfsetarrowsstart{⟨*start arrow kind*⟩}

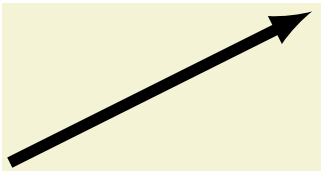
Installs the given ⟨*start arrow kind*⟩ for all subsequent strokes in the in the current TEX-group. If ⟨*start arrow kind*⟩ is empty, no arrow tips will be drawn at the start of paths.



```
\begin{pgfpicture}
  \pgfsetarrowsstart{latex}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetarrowsend{⟨*start arrow kind*⟩}

Like `\pgfsetarrowsstart`, only for the end of the arrow.

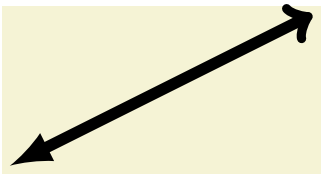


```
\begin{pgfpicture}
  \pgfsetarrowsend{latex}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

Warning: If the compatibility mode is active (which is the default), there also exist old commands called `\pgfsetstartarrow` and `\pgfsetendarrow`, which are incompatible with the meta-arrow management.

\pgfsetarrows{⟨*start kind*⟩-⟨*end kind*⟩}

Calls `\pgfsetarrowsstart` for ⟨*start kind*⟩ and `\pgfsetarrowsend` for ⟨*end kind*⟩.



```
\begin{pgfpicture}
  \pgfsetarrows{latex-to}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{4cm}{2cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

20.5 Predefined Arrow Tip Kinds

The following arrow tip kinds are always defined:

`stealth-stealth`	yields thick ⟷ and thin ⟷
`stealth reversed-stealth reversed`	yields thick ⟷ and thin ⟷
`to-to`	yields thick ⟷ and thin ⟷
`to reversed-to reversed`	yields thick ⟷ and thin ⟷
`latex-latex`	yields thick ⟷ and thin ⟷
`latex reversed-latex reversed`	yields thick ⟷ and thin ⟷
`|-|`	yields thick ⟷ and thin ⟷

For further arrow tips, see page 86.

21 Nodes and Shapes

This section describes the `pgfbaseshapes` package.

```
\usepackage{pgfbaseshapes} % LaTeX
\input pgfbaseshapes.tex   % plain TeX
\input pgfbaseshapes.tex   % ConTeX
```

This package defines commands both for creating nodes and for creating shapes. The package is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

21.1 Overview

PGF comes with a sophisticated set of commands for creating *nodes* and *shapes*. A *node* is a graphical object that consists (typically) of a text label and some additional stroked or filled paths. Each node has a certain *shape*, which may be something simple like a `rectangle` or a `circle`, but it may also be something complicated like a `uml class diagram` (this shape is currently not implemented, though). Different nodes that have the same shape may look quite different, however, since shapes (need not) specify whether the shape path is stroked or filled.

21.1.1 Creating and Referencing Nodes

You create a node by calling the macro `\pgfnode`. This macro takes several parameters and draws the requested shape at a certain position. In addition, it will "remember" the node's position within the current `{pgfpicture}`. You can then, later on, refer to the node's position. Coordinate transformations are "fully supported," which means that if you used coordiante transformations to shift or rotate the shape of a node, the node's position will still be correctly determined by PGF. This is *not* the case if you use canvas transformations, instead.

21.1.2 Anchors

An important property of a node or a shape in general are its *anchors*. Anchors are "important" positions in a shape. For example, the `center` anchor lies at the center of a shape, the `north` anchor is usually "at the top, in the middle" of a shape, the `text` anchor is the lower left corner of the shape's label, and so on.

Anchors are important both when you create a node and when you reference it. When you create a node, you specify the node's "position" by asking PGF to place the shape in such a way that a certain anchor lies at a certain point. For example, you might ask that the node is placed such that the `north` anchor is at the origin. This will effectively cause the node to be placed below the origin.

When you reference a node, you always reference an anchor of the node. For example, when you request the "`north` anchor of the node just placed" you will get the origin. However, you can also request the "`south` anchor of this node," which will give you a point somewhere below the origin. When a coordinate transformation was in force at the time of creation of a node, all anchors are also transformed accordingly.

21.1.3 Layers of a Shape

The simplest shape, the `coordinate`, has just one anchor, namely the `center`, and a label (which is usually empty). More complicated shapes like the `rectangle` shape also have a *background path*. This is a PGF-path that is defined by the shape. The shape does not prescribe what should happen with the path: When a node is created this path may be stroked (resulting in a frame around the label), filled (resulting in a background color for the text), or just discarded.

Although most shapes consist just of a background path plus some label text, when a shape is drawn, up to seven different layers are drawn:

1. The "behind the background layer." Unlike the background path, which be used in different ways by different nodes, these graphic commands given for this layer will always stroke or always fill the path they construct. They might also insert some text that is "behind everything."
2. The background path layer. How this path is used depends on how the arguments of the `\pgfnode` command.

3. The “before the background path layer.” This layer works like the first one, only the commands of this layer are executed after the background path has been used (in whatever way the creator of the node chose).
4. The label layer. This layer inserts the node’s text box.
5. The “behind the foreground layer.” This layer, like the seventh layer, once more contain graphic commands that are “simply executed.”
6. The foreground path layer . This path is treated in the same way as the background path, only it is drawn only after the label text has been drawn.
7. The “before the foreground layer.”

Which of these layers are actually used depends on the shape.

21.2 Creating Nodes

You create a node using the following command:

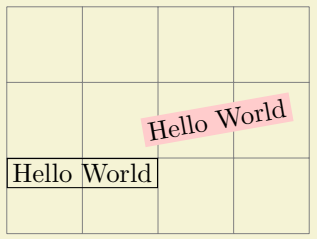
`\pgfnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*label text*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

This command creates a new node. The ⟨*shape*⟩ of the node must have been declared previously using `\pgfdeclareshape`.

The shape is shifted such that the ⟨*anchor*⟩ is at the origin. In order to place the shape somewhere else, use the coordinate transformation prior to calling this command.

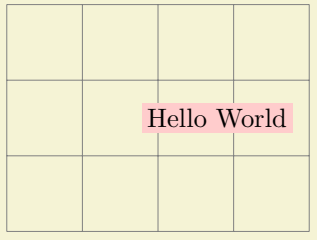
The ⟨*name*⟩ is a name for later reference. If no name is given, nothing will be “saved” for the node, it will just be drawn.

The ⟨*path usage command*⟩ is executed for the background and the foreground path (if the shape defines them).




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \pgftransformshift{\pgfpoint{1cm}{1cm}}
    \pgfnode{rectangle}{north}{Hello World}{hellonode}{\pgfusepath{stroke}}
  }
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

As can be seen, all coordinate transformations are also applied to the text of the shape. Sometimes, it is desirable that the transformations are applied to the point where the shape will be anchored, but you do not wish the shape itself to the transformed. In this case, you should call `\pgftransformresetnontranslations` prior to calling the `\pgfnode` command.



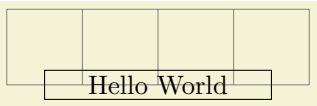

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgftransformresetnontranslations
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

There are a number of values that have an influence on the size of a node. These parameters can be changed using the following commands:

\pgfsetshapeminwidth{⟨*dimension*⟩}

This command sets the macro \pgfshapeminwidth to ⟨*dimension*⟩. This dimension is the *recommended* minimum width of a shape. Thus, when a shape is drawn and when the shape's width would be smaller than ⟨*dimension*⟩, the shape's width is enlarged by adding some empty space.

Note that this value is just a recommendation. A shape may choose to ignore the value of \pgfshapeminwidth altogether.



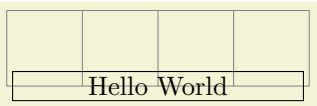

```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeminwidth{3cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

\pgfsetshapeminheight{⟨*dimension*⟩}

Works like \pgfsetshapeminwidth.

\pgfsetshapeinnerxsep{⟨*dimension*⟩}

This command sets the macro \pgfshapeinnerxsep to ⟨*dimension*⟩. This dimension is the *recommended* horizontal inner separation between the label text and the background path. As before, this value is just a recommendation and a shape may choose to ignore the value of \pgfshapeinnerxsep.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeinnerxsep{1cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

\pgfsetshapeinnerysep{⟨*dimension*⟩}

Works like \pgfsetshapeinnerysep.

\pgfsetshapeouterxsep{⟨*dimension*⟩}

This command sets the macro \pgfshapeouterxsep to ⟨*dimension*⟩. This dimension is the recommended horizontal outer separation between the background path and the "outer anchors." For example, if ⟨*dimension*⟩ is `1cm` then the `north` anchor will be 1cm above the top of the background path.

As before, this value is just a recommendation.


```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeouterxsep{.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfsetshapeouterysep{⟨*dimension*⟩}

Works like \pgfsetshapeouterysep.

21.3 Using Anchors

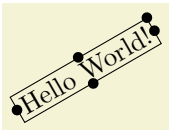
Each shape defines a set of anchors. We saw already that the anchors are used when the shape is drawn: the shape is placed in such a way that the given anchor is at the origin (which in turn is typically translated somewhere else).

One has to look up the set of anchors of each shape, there is no "default" set of anchors, except for the `center` anchor, which should always be present.

Once a node has been defined, you can refer to its anchors using the following commands:

`\pgfpointanchor{`⟨*node*⟩`}{`⟨*anchor*⟩`}`

This command is another "point command" like the commands described in Section 16. It returns the coordinate of the given ⟨*anchor*⟩ in the given ⟨*node*⟩. The command can be used in commands like `\pgfpathmoveto`.




```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

In the above example, you may have noticed something curious: The rotation transformation is still in force when the anchors are invoked, but it does not seem to have an effect. You might expect that the rotation should apply to the already rotated points once more.

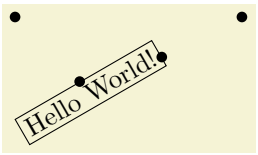
However, `\pgfpointanchor` returns a point that takes the current transformation matrix into account: *The inverse transformation to the current coordinate transformation is applied to anchor point before returning it.*

This behavior may seem a bit strange, but you will find it very natural in most cases. If you really want to apply a transformation to an anchor point (for example, to "shift it away" a little bit), you have to invoke `\pgfpointanchor` without any transformations in force. Here is an example:


```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  {
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
  }

  \pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

`\pgfpointshapeborder{`⟨*node*⟩`}{`⟨*point*⟩`}`

This command returns the point on the border of the shape that lies on a straight line from the center of the node to ⟨*point*⟩. For complex shapes it is not guaranteed that this point will actually lie on the border, it may be on the border of a "simplified" version of the shape.




```
\begin{pgfpicture}
  \begin{pgfscope}
    \pgftransformrotate{30}
    \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  \end{pgfscope}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{-1cm}{1cm}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

21.4 Declaring New Shapes

Defining a shape is, unfortunately, a not-quite-trivial process. The reason is that shapes need to be both very flexible (their size will vary greatly according to circumstances) and they need to be constructed reasonably "fast." PGF must be able to handle pictures with several hundreds of nodes and documents with thousands of nodes in total. It would not do if PGF had to compute and store, say, dozens of anchor positions for the nodes of all pictures.

21.4.1 What Must Be Defined For a Shape?

In order to define a new shape, you must provide:

- a *shape name*,
- code for computing the *saved anchors* and *saved dimensions*,
- code for computing *anchor* positions in terms of the saved anchors,
- optionally code for the *background path* and *foreground path*,
- optionally code for *things to be drawn before or behind* the background and foreground paths.

21.4.2 Normal Anchors Versus Saved Anchors

Anchors are special places in shape. For example, the `north east` anchor, which is a normal anchor, lies at the upper right corner of the `rectangle` shape, as does `\northeast`, which is a saved anchor. The difference is the following: *saved anchors are computed and stored for each node, anchors are only computed as needed.* The user only has access to the normal anchors, but a normal anchor can just "copy" or "pass through" the location of a saved anchor.

The idea behind all this is that a shape can declare a very large number of normal anchors, but when a node of this shape is created, these anchors are not actually computed. However, this causes a problem: When we wish to reference an anchor of a node some time later, we must still able to compute the position of the anchor. For this, we may need a lot of information: What was the transformation matrix that was in force when the node was created? What was the size of the text box? What were the values of the different separation dimensions? And so on.

To solve this problem, PGF will always compute the locations of all *saved anchors* and store these positions. Then, when an normal anchor position is requested later on, the anchor position can be given just from knowing where the locations of the saved anchors.

As an example, consider the `rectangle` shape. For this shape two anchors are saved: The `\northeast` corner and the `\southwest` corner. A normal anchor like `north west` can now easily be expressed in terms of these coordinates: Take the x-position of the `\southwest` point and the y-position of the `\northeast` point. The `rectangle` shape currently defines 13 normal anchors, but needs only two saved anchors. Adding new anchors like a `south south east` anchor would not increase the memory and computation requirements of pictures.

All anchors (both saved and normal) are specified in a local *shape coordinate space*. This is also true for the background and foreground paths. The `\pgfnode` macro will automatically apply appropriate transformations to the coordinates so that the shape is shifted to the right anchor or otherwise transformed.

21.4.3 The Command for Declaring New Shapes

The following command declares a new shape:

`\pgfdeclareshape{`⟨*shape name*⟩`}{`⟨*shape specification*⟩`}`

This command declares a new shape named ⟨*shape name*⟩. The shape name can later be used in commands like `\pgfnode`.

The ⟨*shape specification*⟩ is some TEX code containing calls to special commands that are only defined inside the ⟨*shape specification*⟩ (this is like command like `\draw` that are only available inside the `{tikzpicture}` environment).

Example: Here is the code of the `coordinate` shape:

```
\pgfdeclareshape{coordinate}
{
  \savedanchor\centerpoint{%
    \pgf@x=.5\wd\pgfshapebox%
    \pgf@y=.5\ht\pgfshapebox%
    \advance\pgf@y by -.5\dp\pgfshapebox%
  }
  \anchor{center}{\centerpoint}
  \anchorborder{\centerpoint}
}
```

The special commands are explained next. In the examples given for the special commands a new shape will be constructed, which we might call `simple rectangle`. It should behave like the normal rectangle shape, only without bothering about the fine details like inner and outer separations. The skeleton for the shape is the following.

```
\pgfdeclareshape{simple rectangle}{
  ...
}
```

\savedanchor{⟨*command*⟩}{⟨*code*⟩}

This command declares a saved anchor. ⟨*command*⟩ should be a TEX macro name like `\centerpoint`.

The ⟨*code*⟩ will be executed each time `\pgfnode` is called to create a node of the shape ⟨*shape name*⟩. When the ⟨*code*⟩ is executed, the box `\pgfshapebox` will contain the text label of the node. Possibly, this box is void. The ⟨*code*⟩ can now use the width, height, and depth of the box to compute the location of the saved anchor. In addition, the ⟨*code*⟩ can take into account the valued of dimensions like `\pgfshapeminwidth` or `\pgfshapeinnerxsep`. Furthermore, the ⟨*code*⟩ can take into consideration the values of any further shape-specific variables that are set at the moment when `\pgfnode` is called.

The net effect of the ⟨*code*⟩ should be to set the two TEX dimensions `\pgf@x` and `\pgf@y`. One way to achieve this is to say `\pgfpoint{`⟨*x value*⟩`}{`⟨*y value*⟩`}` at the end of the ⟨*code*⟩, but you can also just set these variables.

The values of `\pgf@x` and `\pgf@y` have after the code has been executed, let us call them x and y, will be recorded and stored together with the node that is created by the command `\pgfnode`.

The macro ⟨*command*⟩ is defined to be `\pgfpoint{`x`}{`y`}`. However, the ⟨*command*⟩ is only locally defined while anchor positions are being computed. Thus, it is possible to use very simple names for ⟨*command*⟩, like `\center` or `\a`, without causing a name-clash. (To be precise, very simple ⟨*command*⟩ names will clash with existing names, but only locally inside the computation of anchor positions; and we do not need the normal `\center` command during these computations.)

For our `simple rectangle` shape, we will need only one saved anchor: The upper right corner. The lower left corner could either be the origin or the "mirrored" upper right corner, depending on whether we want the text label to have its lower left corner at the origin or whether the text label should be centered on the origin. Either will be fine, for the final shape this will make no difference since the shape will be shifted anyway. So, let us assume that the text label is centered on the origin (this will be specified later on using the `text` anchor). We get the following code for the upper right corner:

```
\shapepoint{\upperrightcorner}{
  \pgf@y=.5\ht\pgfshapebox % height of the box, ignoring the depth
  \pgf@x=.5\wd\pgfshapebox % width of the box
}
```

If we wanted to take, say, the `\pgfshapeminwidth` into account, we could use the following code:

```
\shapepoint{\upperrightcorner}{
  \pgf@y=.\ht\pgfshapebox % height of the box
  \pgf@x=.\wd\pgfshapebox % width of the box
  \setlength{\pgf@xa}{\pgfshapeminwidth}
  \ifdim\pgf@x<.5\pgf@xa
    \pgf@x=.5\pgf@xa
  \fi
}
```

Note that we could not have written `.5\pgfshapeminwidth` since the minium width is stored in a "plain text macro," not as a real dimension. So if `\pgfshapeminwidth` depth were 2cm, writing `.5\pgfshapeminwidth` would yield the same as `.52cm`.

In the "real" `rectangle` shape the code is somewhat more complex, but you get the basic idea.

`\saveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command is similar to `\savedanchor`, only instead of setting ⟨*command*⟩ to `\pgfpoint{`x`}{`y`}`, the ⟨*command*⟩ is set just to (the value of) x.

In the `simple rectangle` shape we might use a saved dimension to store the depth of the shape box.

```
\shapedimen{\depth}{
  \pgf@x=\dp\pgfshapebox
}
```

`\anchor{`⟨*name*⟩`}{`⟨*code*⟩`}`

This command declares an anchor named ⟨*name*⟩. Unlike for saved anchors, the ⟨*code*⟩ will not be executed each time a node is declared. Rather, the ⟨*code*⟩ is only executed when the anchor is specifically requested; either for anchoring the node during its creation or as a position in the shape referenced later on.

The ⟨*name*⟩ is a quite arbitrary string that is not "passed down" to the system level. Thus, names like `south` or `1` or `::` would all be fine.

A saved anchor is not automatically also a normal anchor. If you wish to give the users access to a saved anchor you must declare a normal anchor that just returns the position of the saved anchor.

When the ⟨*code*⟩ is executed, all saved anchor macros will be defined. Thus, you can reference them in your ⟨*code*⟩. The effect of the ⟨*code*⟩ should be to set the values of `\pgf@x` and `\pgf@y` to the coordinates of the anchor.

Let us consider some example for the `simple rectangle` shape. First, we would like to make the upper right corner publicly available, for example as `north east`:

```
\anchor{north east}{\upperrightcorner}
```

The `\upperrightcorner` macro will set `\pgf@x` and `\pgf@y` to the coordinates of the upper right corner. Thus, `\pgf@x` and `\pgf@y` will have exactly the right values at the end of the anchor's code.

Next, let us define a `north west` anchor. For this anchor, we can negate the `\pgf@x` variable:

```
\anchor{north west}{
  \upperrightcorner
  \pgf@x=-\pgf@x
}
```

Finally, it is a good idea to always define a `center` anchor, which will be the default location for a shape.

```
\anchor{center}{\pgfpointorigin}
```

You might wonder whether we should not take into consideration that the node is not placed at the origin, but has been shifted somewhere. However, the anchor positions are always specified in the shapes "private" coordinate system. The "outer" transformation that has been applied to the shape upon its creation is applied automatically to the coordinates returned by the anchor's ⟨*code*⟩.

There is one anchor that is special: The `text` anchor. This anchor is used upon creation of a node to determine the lower left corner of the text label (within the private coordinate system of the shape). By default, the `text` anchor is at the origin, but you may change this. For example, we would say

```
\anchor{text}{\pgfpoint{-.5\wd\pgfshapebox}{-.5\ht\pgfshapebox}}
```

to center the text label on the origin in the shape coordinate space.

\anchorborder{⟨*code*⟩}

A *border anchor* is an anchor point on the border of the shape. What exactly is considered as the "border" of the shape depends on the shape.

When the user request a point on the border of the shape using the `\pgfpointshapeborder` command, the ⟨*code*⟩ will be executed to discern this point. When the execution of the ⟨*code*⟩ starts, the dimensions `\pgf@x` and `\pgf@y` will have been set to a location p in the shape's coordinate system. It is now the job of the ⟨*code*⟩ to setup `\pgf@x` and `\pgf@y` such that they specify that point on the shape's border that lies on a straight line from the shape's center to the point p. Usually, this is a somewhat complicated computation, involving many case distinctions and some basic math.

For our `simple rectangle` we must compute a point on the border of a rectangle whose one corner is the origin (ignoring the depth for simplicity) and whose other corner is `\upperrightcorner`. The following code might be used:

```
\anchorborder{%
  % Call a function that computes a border point. Since this
  % function will modify dimensions like \pgf@x, we must move it to
  % other dimensions.
  \@tempdima=\pgf@x
  \@tempdimb=\pgf@y
  \pgfpointborderrectangle{\pgfpoint{\@tempdima}{\@tempdimb}}{\upperrightcorner}
}
```

\backgroundpath{⟨*code*⟩}

This command specifies the path that "makes up" the background of the shape. Note that the shape cannot prescribe what is going to happen with the path: It might be drawn, shaded, filled, or even thrown away. If you want to specify that something should "always" happen when this shape is drawn (for example, if the shape is a stop-sign, we *always* want it to be filled with a red color), you can use commands like `\beforebackgroundpath`, explained below.

When the ⟨*code*⟩ is executed, all saved anchors will be in effect. The ⟨*code*⟩ should contain path construction commands.

For our `simple rectangle`, the following code might be used:

```
\backgroundpath{
  \pgfpathrectanglecorners
    {\upperrightcorner}
    {\pgfpointscale{-1}{\upperrightcorner}}
}
```

As the name suggests, the background path is used "behind" the text label. Thus, this path is used first, then the text label is drawn, possibly obscuring part of the path.

\foregroundpath{⟨*code*⟩}

This command works like `\backgroundpath`, only it is invoked after the text label has been drawn. This means that this path can possibly obscure (part of) the text label.

\behindbackgroundpath{⟨*code*⟩}

Unlike the previous two commands, ⟨*code*⟩ should not only construct a path, it should also use this path in whatever way is appropriate. For example, the ⟨*code*⟩ might fill some area with a unform color.

Whatever the ⟨*code*⟩ does, it does it first. This means that any drawing done by ⟨*code*⟩ will be even behind the background path.

Note that the ⟨*code*⟩ is protected with a `{pgfscope}`.

\beforebackgroundpath{⟨*code*⟩}

This command works like `\behindbackgroundpath`, only the ⟨*code*⟩ is executed after the background path has been used, but before the text label is drawn.

\behindforegroundpath{⟨*code*⟩}

The ⟨*code*⟩ is executed after the text label has been drawn, but before the foreground path is used.

`\beforeforegroundpath{`⟨*code*⟩`}`

This ⟨*code*⟩ is executed at the very end.

`\inheritsavedanchors[from={`⟨*another shape name*⟩`}]`

This command allows you to inherit the code for saved anchors from ⟨*another shape name*⟩. The idea is that if you wish to create a new shape that is just a small modification of a another shape, you can recycle the code used for ⟨*another shape name*⟩.

The effect of this command is the same as if you had called `\savedanchor` and `\saveddimen` for each saved anchor or saved dimension declared in ⟨*another shape name*⟩. Thus, it is not possible to "selectively" inherit only some saved anchors, you always have to inherit all saved anchors from another shape. However, you can inherit the saved anchors of more than one shape by calling this command several times.

`\inheritbehindbackgroundpath[from={`⟨*another shape name*⟩`}]`

This command can be used to inherit the code used for the drawings behind the background path from ⟨*another shape name*⟩.

`\inheritbackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the background path code from ⟨*another shape name*⟩.

`\inheritbeforebackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before background path code from ⟨*another shape name*⟩.

`\inheritbehindforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the behind foreground path code from ⟨*another shape name*⟩.

`\inheritforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the foreground path code from ⟨*another shape name*⟩.

`\inheritbeforeforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before foreground path code from ⟨*another shape name*⟩.

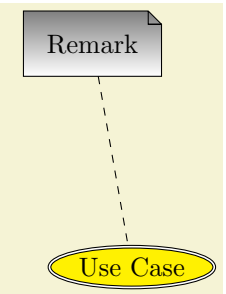
`\inheritanchor[from={`⟨*another shape name*⟩`}]{`⟨*name*⟩`}`

Inherits the code of one specific anchor named ⟨*name*⟩ from ⟨*another shape name*⟩. Thus, unlike saved anchors, which must be inherited collectively, normal anchors can and must be inherited individually.

`\inheritanchorborder[from={`⟨*another shape name*⟩`}]`

Inherits the border anchor code from ⟨*another shape name*⟩.

The following example shows how a shape can be defined that relies heavily on inheritance:



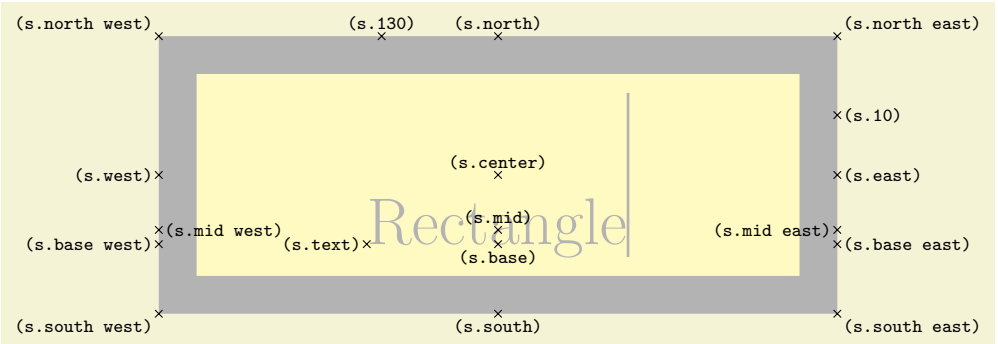

```
\pgfdeclareshape{document}{
  \inheritsavedanchors[from=rectangle] % this nearly a rectangle
  \inheritanchorborder[from=rectangle]
  \inheritanchor[from=rectangle]{center}
  \inheritanchor[from=rectangle]{north}
  \inheritanchor[from=rectangle]{south}
  \inheritanchor[from=rectangle]{west}
  \inheritanchor[from=rectangle]{east}
  % ... and possibly more
  \backgroundpath{% this is new
    % store lower right in xa/ya and upper right in xb/yb
    \southwest \pgf@xa=\pgf@x \pgf@ya=\pgf@y
    \northeast \pgf@xb=\pgf@x \pgf@yb=\pgf@y
    % compute corner of ‘‘flipped page’’
    \pgf@xc=\pgf@xb \advance\pgf@xc by-5pt % this should be a parameter
    \pgf@yc=\pgf@yb \advance\pgf@yc by-5pt
    % construct main path
    \pgfpathmoveto{\pgfpoint{\pgf@xa}{\pgf@ya}}
    \pgfpathlineto{\pgfpoint{\pgf@xa}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@ya}}
    \pgfpathclose
    % add little corner
    \pgfpathmoveto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
 }
}\hskip-1.2cm
\begin{tikzpicture}
  \node[shade,draw,shape=document,inner sep=2ex] (x) {Remark};
  \node[fill=yellow,draw,ellipse,double]
    at ([shift=(-80:3cm)]x) (y) {Use Case};

  \draw[dashed] (x) -- (y);
\end{tikzpicture}
```

21.5 Predefined Shapes

21.5.1 The Rectangle Shape

Shape `rectangle`

This shape is a rectangle tightly fitting the text box. Use inner or outer separation to increase the distance between the text box and the border and the anchors. The following figure shows the anchors this shape defines; the anchors `10` and `130` are example of border anchors.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle,style=shape example] {Rectangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] ((0,0)) node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

21.5.2 The Coordinate Shape

Shape `coordinate`

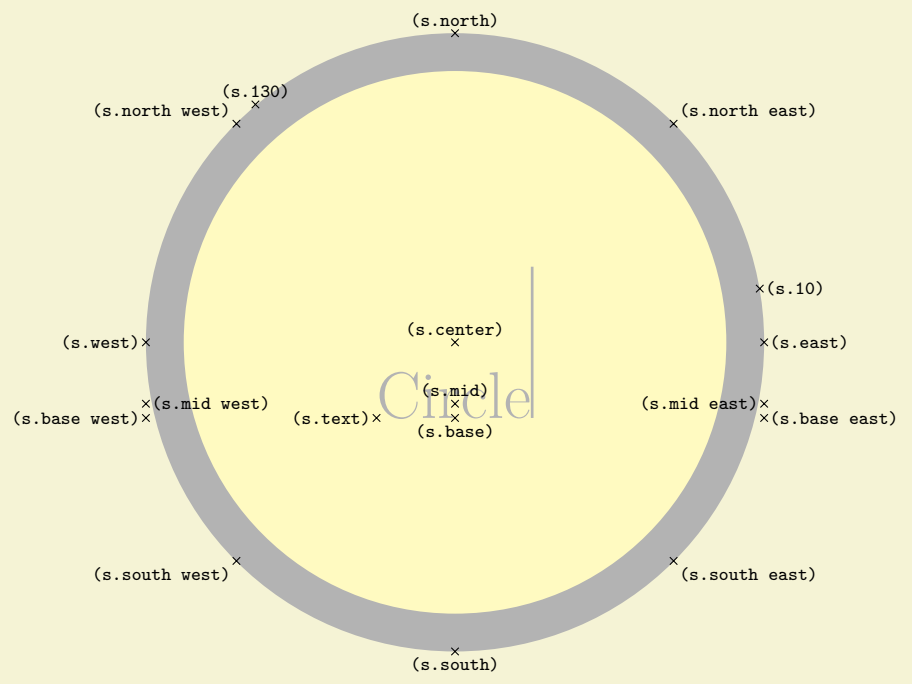
The `coordinate` shape is a special shape that is mainly intended to be used to store locations using the node mechanism. This shape does not have any background path and options like `draw` have no effect on it. If you specify some text, this text will be typeset, but only "a bit unwillingly" since this shape is not really intended for drawing text.

Ti*k*Z handles this shape in a special way, see Section 9.7.

21.5.3 The Circle Shape

Shape `circle`

This shape is a circle tightly fitting the text box.




```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle,style=shape example] {Circle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] ((0,0)) node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

22 Coordinate and Canvas Transformations

22.1 Overview

PGF offers two different ways of scaling, shifting, and rotating (these operations are generally known as *transformations*) or your graphic: You can apply *coordinate transformations* to all coordinates and you can apply *canvas transformations* to the canvas on which you draw. (The names "coordinate" and "canvas" transformations are not standard, they are specially introduced for the purposes of this manual.)

The difference is the following:

- As the name "coordinate transformation" suggests, coordinate transformations apply only to coordinates. For example, if you specify a coordinate like `\pgfpoint{1cm}{2cm}` and you wish to "use" this coordinate—for example as an argument toa `\pgfpathmoveto` command—then the coordinate transformation matrix is applied to the coordinate, resulting in a new coordinate. For example, if the current coordinate transformation is "scale by a factor of two," the coordinate `\pgfpoint{1cm}{2cm}` actually designates the point (2cm, 4cm).

  Note that coordinate transformations apply *only* to coordinates. They do not apply to, say, line width or shadings or text.

- The effect of a "canvas transformation" like "scale by a factor of two" can be imagined as follows: You first draw your picture on a "rubber canvas" normally. Then, once you are done, the whole canvas is transformed, in this case stretched by a factor of two. In the resulting image *everything* will be larger: Text, lines, coordinates, and shadings.

In many cases, it is preferable that you use coordinate transformations and not canvas transformations. When canvas transformations are used, PGF looses track of the coordinates of nodes and shapes. Also, canvas transformations often cause undesirable effects like changing text size. For these reasons, PGF makes it easy to setup the coordinate transformation, but a bit harder to change the canvas transformation.

22.2 Coordinate Transformations

22.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix

PGF has an internal coordinate transformation matrix. This matrix is applied to coordinates "in certain situations." This means that the matrix is not always applied to every coordinate "no matter what." Rather, PGF tries to be reasonably smart at when and how this matrix should be applied. The most prominent examples are the path construction commands, which apply the coordinate transformation matrix to their inputs.

The coordinate transformation matrix consists of four numbers a, b, c, and d, and two dimensions s and t. When the coordinate transformation matrix is applied to a coordinate (x, y) the new coordinate $(ax + by + s, cx + dy + t)$ results. For more details on how transformation matrices work in general, please see, for example, the PDF or PostScript reference or a textbook on computer graphics.

The coordinate transformation matrix is equal to the identity matrix at the beginning. More precisely, $a = 1$, $b = 0$, $c = 0$, $d = 1$, $s = 0\text{pt}$, and $t = 0\text{pt}$.

The different coordinate transformation commands will modify the matrix by concatenating it with another transformation matrix. This way the effect of applying several transformation commands will *accumulate*.

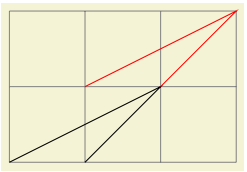
The coordinate transformation matrix is local to the current TEX group (unlike the canvas transformation matrix, which is local to the current `{pgfscope}`). Thus, the effect of adding a coordinate transformation to the coordinate transformation matrix will last only till the end of the current TEX group.

22.2.2 Commands for Relative Coordinate Transformations

The following commands add a basic coordinate transformation to the current coordinate transformation matrix. For all commands, the transformation is applied *in addition* to any previous coordinate transformations.

`\pgftransformshift{`⟨*point*⟩`}`

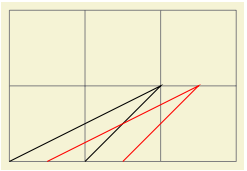
Shifts coordinates by ⟨*point*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformxshift{⟨*dimensions*⟩}

Shifts coordinates by ⟨*dimension*⟩ along the x-axis.



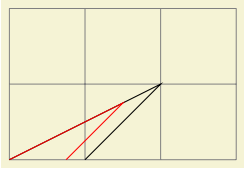
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxshift{.5cm}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyshift{⟨*dimensions*⟩}

Like \pgftransformxshift, only for the y-axis.

\pgftransformscale{⟨*factor*⟩}

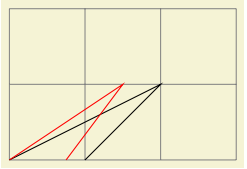
Scales coordinates by ⟨*factor*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformxscale{⟨*factor*⟩}

Scales coordinates by ⟨*factor*⟩ in the x-direction.



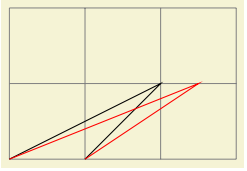
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyscale{⟨*factor*⟩}

Like \pgftransformxscale, only for the y-axis.

\pgftransformxslant{⟨*factor*⟩}

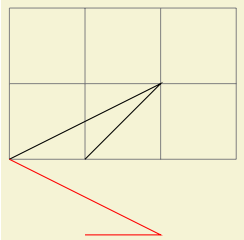
Slants coordinates by ⟨*factor*⟩ in the x-direction. Here, a factor of 1 means 45°.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxslant{.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyslant{⟨*factor*⟩}

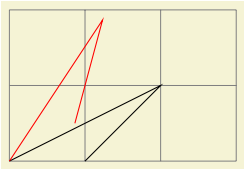
Slants coordinates by ⟨*factor*⟩ in the y-direction.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformyslant{-1}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformrotate{⟨*degrees*⟩}

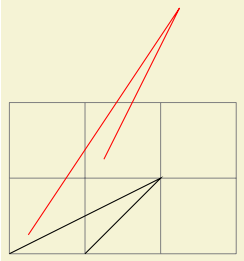
Rotates coordinates counterclockwise by ⟨*degrees*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformrotate{30}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformcm{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*point*⟩}

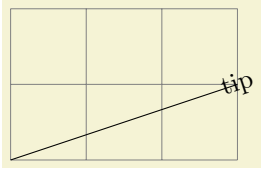
Applies the transformation matrix given by a, b, c, and d and the shift ⟨*point*⟩ to coordinates (in addition to any previous transformations already in force).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformcm{1}{1}{0}{1}{\pgfpoint{.25cm}{.25cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformarrow{⟨*start*⟩}{⟨*end*⟩}

Shift coordinates to the end of the line going from ⟨*start*⟩ to ⟨*end*⟩ with the correct rotation.

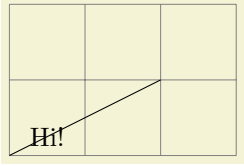



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (3,1);
  \pgftransformarrow{\pgfpointorigin}{\pgfpoint{3cm}{1cm}}
  \pgftext{tip}
\end{tikzpicture}
```

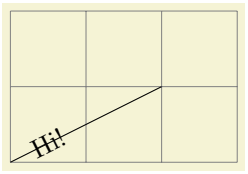
\pgftransformlineattime{⟨*time*⟩}{⟨*start*⟩}{⟨*end*⟩}

Shifts coordinates by a specific point on a line at a specific time. The point by which the coordinate is shifted is calculated by calling `\pgfpointlineattime`, see Section 16.4.2.

In addition to shifting the coordinate, a rotation *may* also be applied. Whether this is the case depends on whether the TEX if `\ifpgfslopedattime` is set to true or not.

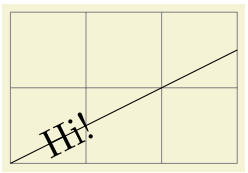



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1);
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

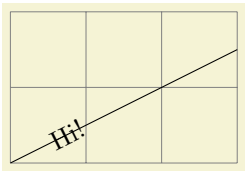



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

There is another TEX if that influences this command. If you set `\ifpgfresetnontranslationattime` to true, then, between shifting the coordinate and (possibly) rotating/sloping the coordinate, the command `\pgftransformresetnontranslations` is called. See the description of this command for details.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimefalse
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

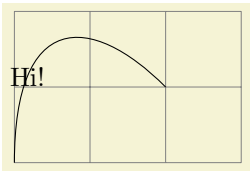



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

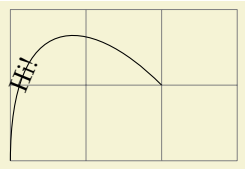
`\pgftransformcurveattime`{⟨*time*⟩}{⟨*start*⟩}{⟨*first support*⟩}{⟨*second support*⟩}{⟨*end*⟩}

Shifts coordinates by a specific point on a curve at a specific time, see Section 16.4.2 once more.

As for the line-at-time transformation command, `\ifpgfslopedattime` decides whether an additional rotation should be applied.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

The value of `\ifpgfresetnontranslationsattime` is also taken into account.

`\ifpgfslopedattime`

Decides whether the "at time" transformation commands also rotate coordinates or not.

`\ifpgfresetnontranslationsattime`

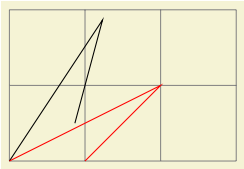
Decides whether the "at time" transformation commands should reset the non-translations between shifting and rotating.

22.2.3 Commands for Absolute Coordinate Transformations

The coordinate transformation commands introduced above are always applied in addition to any previous transformations. In contrast, the commands presented in the following can be used to change the transformation matrix "absolutely." Note that this is, in general, dangerous at will often produce unexpected effects. You should use these commands only if you really know what you are doing.

\pgftransformreset

Resets the coordinate transformation matrix to the identity matrix. Thus, once this command is given no transformations are applied till the end of the scope.



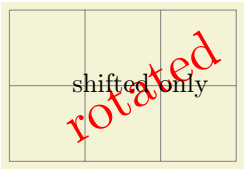
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformreset
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformresetnontranslations

This command sets the a, b, c, and d part of the coordinate transformation matrix to $a = 1$, $b = 0$, $c = 0$, and $d = 1$. However, the current shifting of the matrix is not modified.

The effect of this command is that any rotation/scaling/slanting is undone in the current TEX group, but the origin is not "moved back."

This command is mostly useful directly before a \pgftext command to ensure that the text is not scaled or rotated.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  {\color{red}\pgftext{rotated}}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
\end{tikzpicture}
```

\pgftransforminvert

Replaces the coordinate transformation matrix by a coordinate transformation matrix that "exactly undoes the original transformation." For example, if the original transformation was "scale by 2 and then shift right by 1cm" the new one is "shift left by 1cm and then scale by 1/2."

This command will produce an error if the determinant of the matrix is too small, that is, if the matrix is near-singular.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransforminvert
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

22.2.4 Saving and Restoring the Coordinate Transformation Matrix

There are two commands for saving and later on restoring coordinate transformation matrices.

\pgfgettransform{⟨*macro*⟩}

This command will (locally) define ⟨*macro*⟩ to a representation of the current coordinate transformation matrix. This matrix can later on be reinstalled using \pgfsettransform.

\pgfsettransform{⟨*macro*⟩}

Reinstalls a coordinate transformation matrix that was previously saved using \pgfgettransform.

22.3 Canvas Transformations

The canvas transformation matrix is not managed by PGF, but by the output format like PDF or PostScript. All the PGF does is to call appropriate low-level `\pgfsys@` commands to change the canvas transformation matrix.

Unlike coordinate transformations, canvas transformations apply to "everything," including images, text, shadings, line thickness, and so on. The idea is that a canvas transformation really stretches and deforms the canvas after the graphic is finished.

Unlike coordinate transformations, canvas transformations are local to the current `{pgfscope}`, not to the current TEX group. This is due to the fact that they are managed by the backend driver, not by TEX or PGF.

Unlike the coordinate transformation matrix, it is not possible to "reset" the canvas transformation matrix. The only way to change it is to concatenate it with another canvas transformation matrix or to end the current `{pgfscope}`.

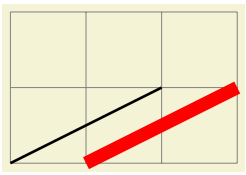
Unlike coordinate transformations, PGF does not "keep track" of canvas transformations. In particular, it will not be able to correctly save the coordinates of shapes or nodes when a canvas transformation is used.

PGF does not offer a whole set of special commands for modifying the canvas transformation matrix. Instead, different commands allow you to concatenate the canvas transformation matrix with a coordinate transformation matrix (and there are numerous commands for specifying a coordinate transformation, see the previous section).

`\pgflowlevelsynccm`

This command concatenates the canvas transformation matrix with the current coordinate transformation matrix. Afterward, the coordinate transformation matrix is reset.

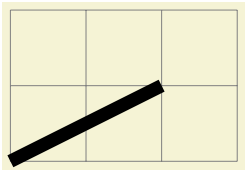
The effect of this command is to "synchronize" the coordinate transformation matrix and the canvas transformation matrix. All transformations the were previously applied by the coordinate transformations matrix are now applied by the canvas transformation matrix.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgftransformscale{5}
  \draw      (0,0) -- (0.4,.2);
  \pgftransformxshift{0.2cm}
  \pgflowlevelsynccm
  \draw[red] (0,0) -- (0.4,.2);
\end{tikzpicture}
```

`\pgflowlevel{`⟨*transformation code*⟩`}`

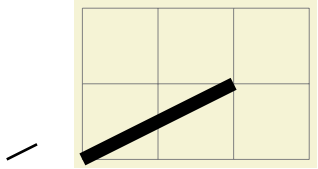
This command concatenates the canvas transformation matrix with the coordinate transformation specified by ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevel{\pgftransformscale{5}}
  \draw      (0,0) -- (0.4,.2);
\end{tikzpicture}
```

`\pgflowlevelobj{`⟨*transformation code*⟩`}{`⟨*code*⟩`}`

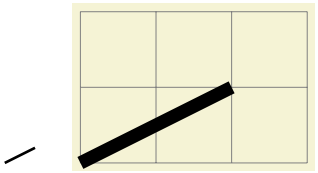
This command creates a local `{pgfscope}`. Inside this scope, `\pgflowlevel` is first called with the argument ⟨*transformation code*⟩, then the ⟨*code*⟩ is inserted.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevelobj{\pgftransformscale{5}}    {\draw (0,0) -- (0.4,.2);}
  \pgflowlevelobj{\pgftransformxshift{-1cm}}{\draw (0,0) -- (0.4,.2);}
\end{tikzpicture}
```

`\begin{pgflowlevelscope}{`⟨*transformation code*⟩`}`
⟨*environment contents*⟩
`\end{pgflowlevelscope}`

This environment first surrounds the ⟨*environment contents*⟩ by a `{pgfscope}`. Then it calls `\pgflowlevel` with the argument ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \begin{pgflowlevelscope}{\pgftransformscale{5}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
  \begin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
\end{tikzpicture}
```

23 Declaring and Using Shadings

23.1 Overview

A shading is an area in which the color changes smoothly between different colors. Note also that `ghostview` may do a poor job at displaying shadings when doing anti-aliasing.

Similarly to an image, a shading must first be declared before it can be used. Also similarly to an image, a shading is put into a TEX-box. Hence, in order to include a shading in a `{pgfpicture}`, you have to use `\pgftext` around it.

There are three kinds of shadings: horizontal, vertical, and radial shadings. However, you can rotate and clip shadings like any other graphics object, which allows you to create more complicated shadings. Horizontal shadings could be created by rotating a vertical shading by 90 degrees, but explicit commands for creating both horizontal and vertical shadings are included for convenience.

Once you have declared a shading, you can insert it into text using the command `\pgfuseshading`. This command cannot be used directly in a `{pgfpicture}`, you have to put a `\pgftext` around it. The second command for using shadings, `\pgfshadepath`, on the other hand, can only be used inside `{pgfpicture}` environments. It will "fill" the current path with the shading.

A horizontal shading is a horizontal bar of a certain height whose color changes smoothly. You must at least specify the colors at the left and at the right end of the bar, but you can also add color specifications for points inbetween. For example, suppose you which to create a bar that is red at the left end, green in the middle, and blue at the end. Suppose you would like the bar to be 4cm long. This could be specified as follows:

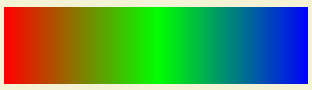
```
rgb(0cm)=(1,0,0); rgb(2cm)=(0,1,0); rgb(4cm)=(0,0,1)
```

This line means that at 0cm (the left end) of the bar, the color should be red, which has red-green-blue (rgb) components (1,0,0). At 2cm, the bar should be green, and at 4cm it should be blue. Instead of `rgb`, you can currently also specify `gray` as color model, in which case only one value is needed, or `color`, in which case you must provide the name of a color in round brackets. In a color specification the individual specifications must be separated using a semicolon, which may be followed by a whitespace (like a space or a newline). Individual specifications must be given in increasing order.

23.2 Declaring Shadings

`\pgfdeclarehorizontalshading[`⟨*color list*⟩`]{`⟨*shading name*⟩`}{`⟨*shading height*⟩`}{`⟨*color specification*⟩`}`

Declares a horizontal shading named ⟨*shading name*⟩ of the specified ⟨*height*⟩ with the specified colors. The length of the bar is automatically deduced from the maximum specification.



```
\pgfdeclarehorizontalshading{myshading}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(green); color(4cm)=(blue)}
\pgfuseshading{myshading}
```

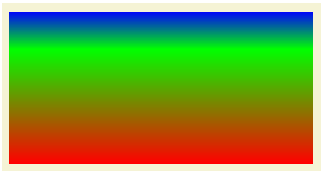
The effect of the ⟨*color list*⟩, which is a comma-separated list of colors, is the following: Normally, when this list is empty, once a shading is declared it becomes "frozen." This means that even if you change a color that was used in the declaration of the shading later on, the shading will not change. By specifying a ⟨*color list*⟩ you can specify that the shading should be recalculated whenever one of the colors listed in the list changes (this includes effects like color mixins). Thus, when you specify a ⟨*color list*⟩, whenever the shading is used, PGF first converts the colors in the list to RGB triples using the current values of the colors and taking any mixins and blendings into account. If the resulting RGB triples have not yet been used, a new shading is internally created and used. Note that if the option ⟨*color list*⟩ is used, then no shading is created until the first use of `\pgfuseshading`. In particular, the colors mentioned in the shading need not be defined when the declaration is given.

When a shading is recalculated because of a change in the colors mentioned in ⟨*color list*⟩, the complete shading is recalculated. Thus even colors not mentioned in the list will be used with their current values, not with the values they had upon declaration.

```
\pgfdeclarehorizontalshading[mycolor]{myshading}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(mycolor)}
\colorlet{mycolor}{green}
\pgfuseshading{myshading}
\colorlet{mycolor}{blue}
\pgfuseshading{myshading}
```

\pgfdeclareverticalshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*shading width*⟩}{⟨*color specification*⟩}

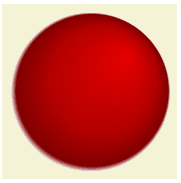
Declares a vertical shading named ⟨*shading name*⟩ of the specified ⟨*width*⟩. The height of the bar is automatically deduced from the maximum specification. The effect of color list is the same as for horizontal shadings.



```
\pgfdeclareverticalshading{myshading2}
  {4cm}{rgb(0cm)=(1,0,0); rgb(1.5cm)=(0,1,0); rgb(2cm)=(0,0,1)}
\pgfuseshading{myshading2}
```

\pgfdeclareradialshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*center point*⟩}{⟨*color specification*⟩}

Declares an radial shading. A radial shading is a circle whose inner color changes as specified by the color specification. Assuming that the center of the shading is at the origin, the color of the center will be the color specified for 0cm and the color of the border of the circle will be the color for the maximum specification. The radius of the circle will be the maximum specification. If the center coordinate is not at the origin, the whole shading inside the circle (whose size remains exactly the same) will be distorted such that the given center now has the color specified for 0cm. The effect of color list is the same as for horizontal shadings.



```
\pgfdeclareradialshading{sphere}{\pgfpoint{0.5cm}{0.5cm}}%
  {rgb(0cm)=(0.9,0,0);
   rgb(0.7cm)=(0.7,0,0);
   rgb(1cm)=(0.5,0,0);
   rgb(1.05cm)=(1,1,1)}
\pgfuseshading{sphere}
```

\pgfaliasshading{⟨*new shading name*⟩}{⟨*existing shading name*⟩}

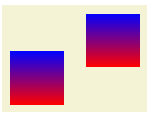
The ⟨*existing shading name*⟩ is "cloned" and the shading ⟨*new shading name*⟩ can now be used whenever original shading is used. This command is mainly useful for creating aliases for environments that use alternate extensions.

Example: `\pgfaliasshading{shading!30}{shading!25}`

23.3 Using Shadings

\pgfuseshading{⟨*shading name*⟩}

Inserts a previously declared shading into the text. If you wish to use it in a `pgfpicture` environment, you should put a `\pgfbox` around it. Like `\pgfuseimage`, alternate extensions are tried before the actual shading is used.



```
\begin{pgfpicture}
  \pgfdeclareverticalshading{shading}
    {20pt}{color(0pt)=(red); color(20pt)=(blue)}
  \pgftext[at=\pgfpoint{1cm}{0cm}]   {\pgfuseshading{shading}}
  \pgftext[at=\pgfpoint{2cm}{0.5cm}]{\pgfuseshading{shading}}
\end{pgfpicture}
```

\pgfshadepath{⟨*shading name*⟩}{⟨*angle*⟩}

This command must be used inside a `{pgfpicture}` environment. The effect is a bit complex, so let us go over it step by step.

First, PGF will setup a local scope.

Second, it uses the current path to clip everything inside this scope. However, the current path is once more available after the scope, so it can be used, for example, to stroke it.

Now, the ⟨*shading name*⟩ should be a shading whose width and height are 100 bp, that is, 100 big points. PGF has a look at the bounding box of the current path. This bounding box is computed

automatically when a path is computed; however, it can sometimes be (quite a bit) too large, especially when complicated curves are involved.

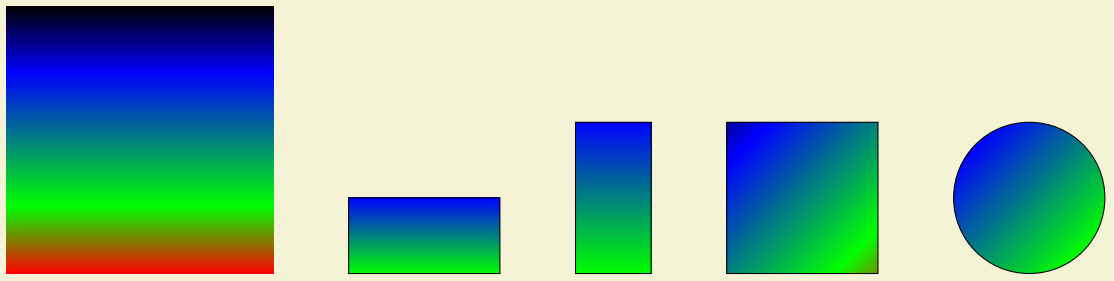
Inside the scope, the lowlevel transformation matrix is modified. The center of the shading is translated (moved) such that it lies on the center of the bounding box of the path. The lowlevel coordinate system is also scaled such that the shading "covers" the shading (the details are a bit more complex, see below). Finally, the coordinate system is rotated by ⟨*angle*⟩.

After everything has been set up, the shading is inserted. Due to the transformations and clippings, the effect will be that the shading seems to "fill" the path.

If both the path and the shadings were always be rectangles and if rotation were never involved, it would be easy to scale shadings such they always cover the path. However, when a vertical shading is rotated, must must obviously "magnify" the shading so that it still covers the path. Things get worse when the path is not a rectangle itself.

For these reasons, things work slightly differently "in reality." The shading is scaled (more precisely, the coordinate system is scaled, but never mind the difference) and translated such that the the point (50bp, 50bp), which is the middle of the shading, is at the middle of the path and such that the the point (25bp, 25bp) is at the lower left corner of the path and that (75bp, 75bp) is at upper right corner.

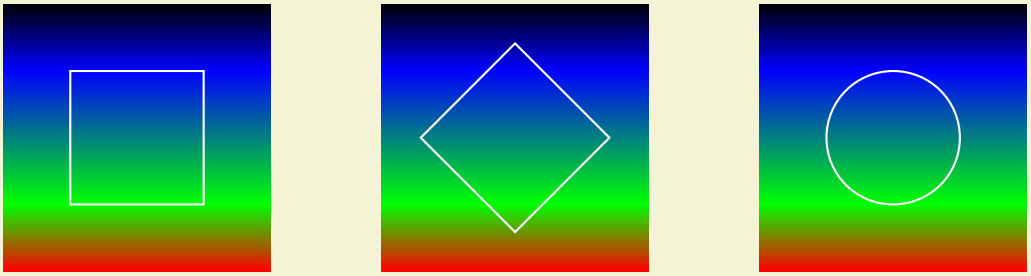
In other words, only the center quarter of the shading will actually "survive the clipping" if the path is a rectangle. If it is not a rectangle, but, say, a circle, even less is seen of the shading. Here is an example that demonstrates this effect:



```
\pgfdeclareverticalshading{shading}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\pgfuseshading{shading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{shading}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{1cm}{2cm}}
  \pgfshadepath{shading}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{5cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfshadepath{shading}{45}
  \pgfusepath{stroke}
  \pgfpathcircle{\pgfpoint{9cm}{1cm}}{1cm}
  \pgfshadepath{shading}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

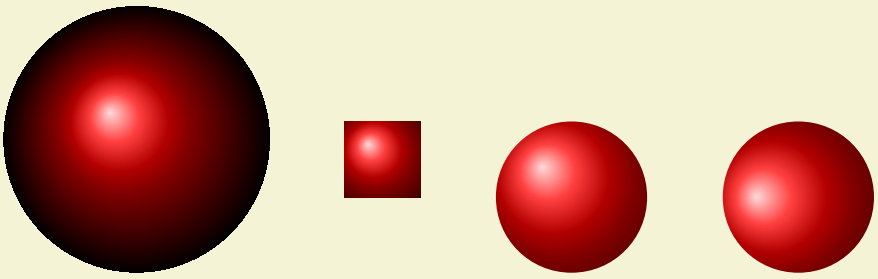
As can be seen above in the last case, the "hidden" part of the shading actually *can* become visible if the shading is rotated. The reason is that it is scaled as if no rotation took place, then the rotation is done.

The following graphics show which part of the shading are actually shown:




```
\begin{tikzpicture}
  \draw (50bp,50bp) node {\pgfuseshading{shading}};
  \draw[white,thick] (25bp,25bp) rectangle (75bp,75bp);
  \draw (50bp,0bp) node[below] {first two applications};

  \begin{scope}[xshift=5cm]
    \draw (50bp,50bp) node{\pgfuseshading{shading}};
    \draw[rotate around={45:(50bp,50bp)},white,thick] (25bp,25bp) rectangle (75bp,75bp);
    \draw (50bp,0bp) node[below] {third application};
  \end{scope}

  \begin{scope}[xshift=10cm]
    \draw (50bp,50bp) node{\pgfuseshading{shading}};
    \draw[white,thick] (50bp,50bp) circle (25bp);
    \draw (50bp,0bp) node[below] {fourth application};
  \end{scope}
\end{tikzpicture}
```

An advantage of this approach is that when you rotate a radial shading, no distortion is introduced:

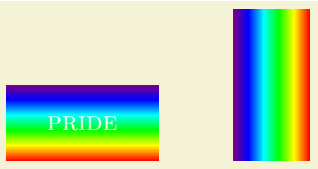


```
\pgfdeclareradialshading{ballshading}{\pgfpoint{-10bp}{10bp}}
 {color(0bp)=(red!15!white); color(9bp)=(red!75!white);
 color(18bp)=(red!70!black); color(25bp)=(red!50!black); color(50bp)=(black)}
\pgfuseshading{ballshading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{3cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{6cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{45}
  \pgfusepath{}
\end{pgfpicture}
```

If you specify a rotation of 90° and if the path is not a square, but an elongated rectangle, the “desired” effect results: The shading will exactly vary between the colors at the 25bp and 75bp boundaries. Here is an example:

```
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{shading}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{shading}{90}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{6cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{shading}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

As a final example, let us define a "rainbow spectrum" shading for use with Ti*k*Z.




```
\pgfdeclareverticalshading{rainbow}{100bp}
 {color(0bp)=(red); color(25bp)=(red); color(35bp)=(yellow);
  color(45bp)=(green); color(55bp)=(cyan); color(65bp)=(blue);
  color(75bp)=(violet); color(100bp)=(violet)}
\begin{tikzpicture}[shading=rainbow]
  \shade (0,0) rectangle node[white] {\textsc{pride}} (2,1);
  \shade[shading angle=90] (3,0) rectangle +(1,2);
\end{tikzpicture}
```

Note that rainbow shadings are *way* to colorful in almost all applications.

24 Declaring and Using Images

This section describes the `pgfbaseimage` package.

```
\usepackage{pgfbaseimage} % LaTeX
\input pgfbaseimage.tex   % plain TeX
\input pgfbaseimage.tex   % ConTeX
```

This package offers an abstraction of the image inclusion process. It is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

24.1 Overview

To be quite frank, LaTeX's `\includegraphics` is designed better than `pgfbaseimage`. For this reason, *I recommend that you use the standard image inclusion mechanism of your format.* Thus, LaTeX users are encouraged to use `\includegraphics` to include images.

However, there are reasons why you might need to use the image inclusion facilities of PGF:

- There is no standard image inclusion mechanism in your format. For example, plain TeX does not have one, so PGF's inclusion mechanism is "better than nothing."

  However, this applies only to the `pdftex` backend. For all other backends, PGF just maps its command back to the `graphicx` package. Thus, in plain TeX, this does not really help. It might be a good idea to fix this in the future such that PGF becomes independent of LaTeX, thereby providing a uniform image abstraction for all formats.

- You wish to use masking. This is a feature that is only supported by PGF, though I hope that someone will implement this also for the graphics package in LaTeX in the future.

Whatever you choice, you can still use the usual image inclusion facilities of the `graphics` package.

The general approach taken by PGF to including an image is the following: First, `\pgfdeclareimage` declares the image. This must be done prior to the first use of the image. Once you have declared an image, you can insert it into the text using `\pgfuseimage`. The advantage of this two-phase approach is that, at least for PDF, the image data will only be included once in the file. This can drastically reduce the file size if you use an image repeatedly, for example in an overlay. However, there is also a command called `\pgfimage` that declares and then immediately uses the image.

To speedup the compilation, you may wish to use the following class option:

```
\usepackage[draft]{pgf}
```

In draft mode boxes showing the image name replace the images. It is checked whether the image files exist, but they are not read. If either height or width is not given, 1cm is used instead.

24.2 Declaring an Image

`\pgfdeclareimage[`⟨*options*⟩`]{`⟨*image name*⟩`}{`⟨*filename*⟩`}`

Declares an image, but does not paint anything. To draw the image, use `\pgfuseimage{`⟨*image name*⟩`}`. The ⟨*filename*⟩ may not have an extension. For PDF, the extensions `.pdf`, `.jpg`, and `.png` will automatically tried. For PostScript, the extensions `.eps`, `.epsi`, and `.ps` will be tried.

The following options are possible:

- `height=`⟨*dimension*⟩ sets the height of the image. If the width is not specified simultaneously, the aspect ratio of the image is kept.
- `width=`⟨*dimension*⟩ sets the width of the image. If the height is not specified simultaneously, the aspect ratio of the image is kept.
- `page=`⟨*page number*⟩ selects a given page number from a multipage document. Specifying this option will have the following effect: first, PGF tries to find a file named

  ⟨*filename*⟩`.page`⟨*page number*⟩`.`⟨*extension*⟩

  If such a file is found, it will be used instead of the originally specified filename. If not, PGF inserts the image stored in ⟨*filename*⟩`.`⟨*extension*⟩ and if a recent version of `pdflatex` is used, only the selected page is inserted. For older versions of `pdflatex` and for `dvips` the complete document is inserted and a warning is printed.

- `interpolate=`⟨*true or false*⟩ selects whether the image should "smoothed" when zoomed. False by default.
- `mask=`⟨*mask name*⟩ selects a transparency mask. The mask must previously be declared using `\pgfdeclaremask` (see below). This option only has an effect for `pdf`. Not all viewers support masking.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1}{pgf-tu-logo}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2}{pgf-tu-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{pgf-tu-logo}
```

`\pgfaliasimage{`⟨*new image name*⟩`}{`⟨*existing image name*⟩`}`

The `{`⟨*existing image name*⟩`}` is "cloned" and the `{`⟨*new image name*⟩`}` can now be used whenever original image is used. This command is useful for creating aliases for alternate extensions and for accessing the last image inserted using `\pgfimage`.

Example: `\pgfaliasimage{image.!30!white}{image.!25!white}`

24.3 Using an Image

`\pgfuseimage{`⟨*image name*⟩`}`

Inserts a previously declared image into the *normal text*. If you wish to use it in a `{pgfpicture}` environment, you must put a `\pgftext` around it.

If the macro `\pgfalternateextension` expands to some nonempty ⟨*alternate extension*⟩, PGF will first try to use the image names ⟨*image name*⟩`.`⟨*alternate extension*⟩. If this image is not defined, PGF will next check whether ⟨*alternate extension*⟩ contains a `!` character. If so, everythings up to this exclamation mark and including it is deleted from ⟨*alternate extension*⟩ and the PGF again tries to use the image ⟨*image name*⟩`.`⟨*alternate extension*⟩. This is repeated until ⟨*alternate extension*⟩ no longer contains a `!`. Then the original image is used.

The `xxcolor` package sets the alternate extension to the current color mixin.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1}{pgf-tu-logo}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2}{pgf-tu-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{pgf-tu-logo}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{3cm}{1cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{5cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{3cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{5cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following example demonstrates the effect of using `\pgfuseimage` inside a color mixin environment.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1.!25!white}{pgf-tu-logo.25}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2.25!white}{pgf-tu-logo.25}
\pgfdeclareimage[interpolate=true,height=1cm]{image3.white}{pgf-tu-logo.25}
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{3cm}{1cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{5cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{3cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{5cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

\pgfalternateextension

You should redefine this command to install a different alternate extension.

Example: `\def\pgfalternateextension{!25!white}`

\pgfimage[⟨*options*⟩]{⟨*filename*⟩}

Declares the image under the name `pgflastimage` and immediately uses it. You can "save" the image for later usage by invoking `\pgfaliasimage` on `pgflastimage`.

```
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]
    {\pgfimage[interpolate=true,width=1cm,height=1cm]{pgf-tu-logo}}
  \pgftext[at=\pgfpoint{3cm}{1cm},left,base]
    {\pgfimage[interpolate=true,width=1cm]{pgf-tu-logo}}
  \pgftext[at=\pgfpoint{5cm}{1cm},left,base]
    {\pgfimage[interpolate=true,height=1cm]{pgf-tu-logo}}

  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{3cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{5cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

24.4 Masking an Image

\pgfdeclaremask[⟨*options*⟩]{⟨*mask name*⟩}{⟨*filename*⟩}

Declares a transparency mask named ⟨*mask name*⟩ (called a *soft mask* in the PDF specification). This mask is read from the file ⟨*filename*⟩. This file should contain a grayscale image that is as large as the actual image. A white pixel in the mask will correspond to "transparent," a black pixel to "solid," and grey values correspond to intermediate values. The mask must have a single "color channel." This means that the mask must be a "real" grayscale image, not an RGB-image in which all RGB-triples happen to have the same components.

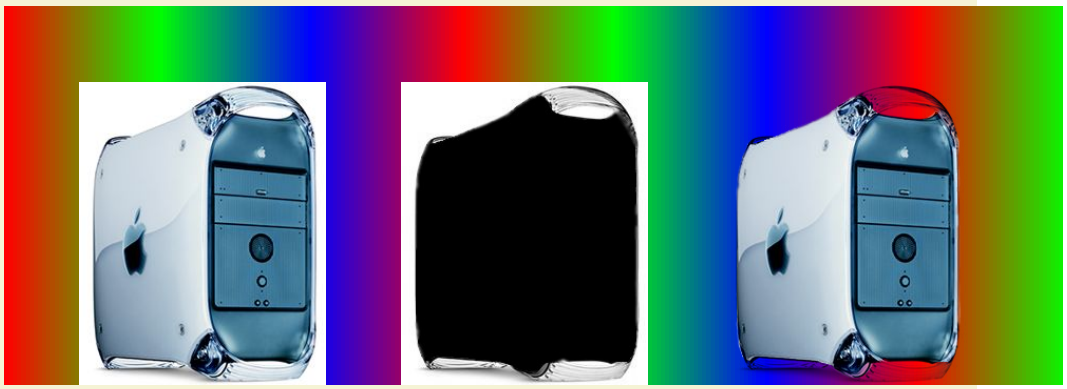
You can only mask images the are in a "pixel format." These are `.jpg` and `.png`. You cannot mask `.pdf` images in this way. Also, again, the mask file and the image file have to have the same size.

The following options may be given:

- `matte={`⟨*color components*⟩`}` sets the so-called *matte* of the actual image (strangely, this has to be specified together with the mask, not with the image itself). The matte is the color that has been used to preblend the image. For example, if the image has been preblended with a red background, then ⟨*color components*⟩ should be set to `{1 0 0}`. The default is `{1 1 1}`, which is white in the rgb model.

  The matte is specified in terms of the parent's image color space. Thus, if the parent is a grayscale image, the matte has to be set to `{1}`.

Example:



```
%% Draw a large colorful background
\pgfdeclarehorizontalshading{colorful}{5cm}{color(0cm)=(red);
color(2cm)=(green); color(4cm)=(blue); color(6cm)=(red);
color(8cm)=(green); color(10cm)=(blue); color(12cm)=(red);
color(14cm)=(green)}
\hbox{\pgfuseshading{colorful}\hskip-14cm\hskip1cm
\pgfimage[height=4cm]{pgf-apple}\hskip1cm
\pgfimage[height=4cm]{pgf-apple.mask}\hskip1cm
\pgfdeclaremask{mymask}{pgf-apple.mask}
\pgfimage[mask=mymask,height=4cm,interpolate=true]{pgf-apple}}
```

25 Creating Plots

This section describes the pgfbaseplot package.

```
\usepackage{pgfbaseplot} % LaTeX
\input pgfbaseplot.tex   % plain TeX
\input pgfbaseplot.tex   % ConTeX
```

This package provides a set of commands that are intended to make it reasonably easy to plot functions using PGF. It is loaded automatically by pgf, but you can load it manually if you have only included pgfcore.

25.1 Overview

25.1.1 When Should One Use PGF for Generating Plots?

There exist many powerful programs that produce plots, examples are GNUPLOT or MATHEMATICA. These programs can produce two different kinds of output: First, they can output a complete plot picture in a certain format (like PDF) that includes all lowlevel commands necessary for drawing the complete plot (including axes and labels). Second, they can usually also produce "just plain data" in the form of a long list of coordinates. Most of the powerful programs consider it a to be "a bit boring" to just output tabled data and very much prefer to produce fancy pictures. Nevertheless, when coaxed, they can also provide the plain data.

The plotting mechanism described in the following deals only with plotting data given in the form of a list of coordinates. Thus, this section is about using PGF to turn lists of coordinates into plots.

Note that is often not necessary to use PGF *for this.* Programs like GNUPLOT can produce very sophisticated plots and it is usually much easier to simply include these plots as a finished PDF or PostScript graphics.

However, there are a number of reasons why you may wish to invest time and energy into mastering the PGF commands for creating plots:

- Virtually all plots produced by "external programs" use different fonts from the one used in your document.
- Even worse, formulas will look totallly different, if they can be rendered at all.
- Line width will usually be too large or too small.
- Scaling effects upon inclusiong can create a mismatch between sizes in the plot and sizes in the text.
- The automatic grid generated by most programs is mostly distracting.
- The automatic ticks generated by most programs are cryptic numerics. (Try adding a tick reading π at the right point.)
- Most programs make it very easy to create "chart junk" in a most convenient fashion. All show, no content.
- Arrows and plot marks will almost never match the arrows used in the rest of the document.

The above list is not exhaustive, unfortunately.

25.1.2 How PGF Handles Plots

PGF (conceptually) uses a two-stage process for generating plots. First, a *plot stream* must be produced. This stream consists (more or less) of a large number of coordinates. Second a *plot handler* is applied to the stream. A plot handler "does something" with the coordinates. The standard handler will issue line-to operations to the coordinates in the stream. However, a handler might also try to issue appropriate curve-to operations in order to smooth the curve. A handler may even do something else entirely, like writing each coordinate to another stream, thereby duplicating the original stream.

Both for the creating of streams and the handling of streams different sets of commands exist. The commands for creating streams start with \pgfplotstream, the commands for setting the handler start with \pgfplothandler.

25.2 Generating Plot Streams

25.2.1 Basic Building Blocks of Plot Streams

A *plot stream* is a (long) sequence of the following three commands:

1. `\pgfplotstreamstart`,
2. `\pgfplotstreampoint`, and
3. `\pgfplotstreamend`.

Between calls of these commands arbitrary other code may be called. Obviously, the stream should start with the first command and end with the last command. Here is an example of a plot stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\newdimen\mydim
\mydim=2cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\advance \mydim by 3cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreamend
```

`\pgfplotstreamstart`

This command signals that a plot stream starts. The effect of this command is to call the internal command `\pgf@plotstreamstart`, which is set by the current plot handler to do whatever needs to be done at the beginning of the plot.

`\pgfplotstreampoint{`⟨*point*⟩`}`

This command adds a ⟨*point*⟩ to the current plot stream. The effect of this command is to call the internal command `\pgf@plotstreampoint`, which is also set by the current plot handler. This command should now "handle" the point in some sensible way. For example, a line-to command might be issued for the point.

`\pgfplotstreamend`

This command signals that a plot stream ends. It calls `\pgf@plotstreamend`, which should now do any necessary "cleanup."

Note that plot streams are not buffered, that is, the different points are immediately handled. However, using the recording handler, it is possible to record a stream.

25.2.2 Commands the Generate Plot Streams

Plot streams be created "by hand" as in the earlier example. However, most of the time the coordinates will be produced by some command. For example, the `\pgfplotxyfile` reads a file and converts it into a plot stream.

`\pgfplotxyfile{`⟨*filename*⟩`}`

This command will try to open the file ⟨*filename*⟩. If this succeeds, it will convert the file contents into a plot stream as follows: A `\pgfplotstreamstart` is issued (how would have thought...). Then, each nonempty line of the file should start with two numbers separated by a space, such as `0.1 1` or `100 -.3`. Anything following the numbers is ignored.

Each pair ⟨x⟩ and ⟨y⟩ of numbers is converted into one plot stream point in the xy-coordinate system. Thus, a line like

```
2 -5 some text
```

is turned into

```
\pgfplotstreampoint{\pgfpointxy{2}{-5}}
```

The two characters `%` and `#` are also allowed in a file and they are both treated as command characters. Thus, a line starting with either of them is empty and, hence, ignored.

When the file has been read completely, `\pgfplotstreamend` is called.

\pgfplotxyzfile{⟨*filename*⟩}

This command works like \pgfplotxyfile, only *three* numbers are expected on each non-empty line. They are converted into points in the xyz-coordinate system. Consider, the following file:

```
% Some comments
# more comments
2 -5  1 first entry
2 -.2 2 second entry
2 -5  2 third entry
```

is turned into

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{1}}
\pgfplotstreampoint{\pgfpointxyz{2}{-.2}{2}}
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{2}}
\pgfplotstreamend
```

Currently, there is no command that can decide automatically whether the xy-coordinate system should be used or whether the xyz-system should be used. However, it would not be terribly difficult to write a "smart file reader" that parses coordinate files a bit more intelligently.

\pgfplotgnuplot[⟨*prefix*⟩]{⟨*function*⟩}

This command will "try" to call the GNUPLOT program to generate the coordinates of the ⟨*function*⟩. In detail, the following happens:

This command will works with two files: ⟨*prefix*⟩.gnuplot and ⟨*prefix*⟩.table. If the optional argument ⟨*perfix*⟩ is not given, it is set to \jobname.

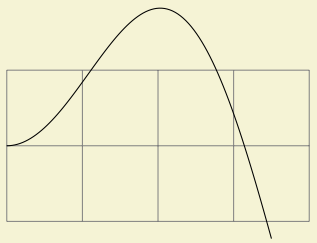
Let us start with the situation, that none of these files exists. Then, PGF will first generate the file ⟨*prefix*⟩.gnuplot. In this file, it will first write

```
set terminal table; set output "#1.table"; set format "%.5f"
```

where #1 is replaced by ⟨*prefix*⟩. Then, in a second line, it writes the text ⟨*function*⟩.

Next, PGF will try to invoke the program gnuplot with the argument ⟨*prefix*⟩.gnuplot. This call may or may not succeed, depending on whether the \write18 mechanism (also known as shell-esapce) is switched on and whether the gnuplot program is available.

Assuming that the call succeeded, the next step is to invoke \pgfplotxyfile on the file ⟨*prefix*⟩.table; which is exactly the file that has just been created by gnuplot.



```
\begin{tikzpicture}
  \draw[help lines] (0,-1) grid (4,1);
  \pgfplothandlerlineto
  \pgfplotgnuplot[plots/pgfplotgnuplot-example]{plot [x=0:3.5] x*sin(x)}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The more difficult situation arises when the .gnuplot file exists, which will be the case on the second run of TEX on the TEX file. In this case, PGF will read this file and check whether it contains exactly the could that is "would have written" into this file. If this is not the case, the file contents is overwritten with what "should be there" and, as above, gnuplot is invoked to generate a new .table file. However, if the file contents is "as expected," the external gnuplot program is *not* called. Instead, the ⟨*prefix*⟩.table file is immediately read.

As explained in Section 7.13.3, the net effect of the above mechanism is that gnuplot is called as little as possible and that when you pass along the .gnuplot and .table files with your .tex file to someone else, that person can TEX the .tex file without having gnuplot installed and without having the \write18 mechanism switched on.

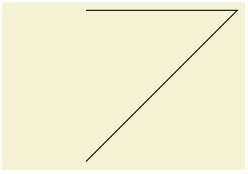
25.3 Plot Handlers

A *plot handler* prescribes what "should be done" with a plot stream. You must set the plot handler before the stream starts. The following commands install the most basic plot handlers; more plot handlers are defined in the file `pgflibraryplothandlers`, which is documented in Section 11.2.

All plot handlers work by setting redefining the following three macros: `\pgf@plotstreamstart`, `\pgf@plotstreampoint`, and `\pgf@plotstreamend`.

`\pgfplothandlerlineto`

This handler will issue a `\pgfpathlineto` command for each point of the plot, *except* possibly for the first. What happens with the first point can be specified using the two commands described below.



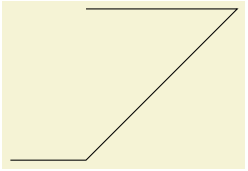
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetmovetofirstplotpoint`

Specifies that the line-to plot handler (and also some other plot handlers) should issue a move-to command for the first point of the plot instead of a line-to. This will start a new part of the current path, which is not always, but often, desirable. This is the default.

`\pgfsetlinetofirstplotpoint`

Specifies that plot handlers should issue a line-to command for the first point of the plot.



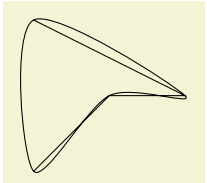
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfsetlinetofirstplotpoint
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfplothandlerdiscard`

This handler will simply throw away the stream.

`\pgfplothandlerrecord{`⟨*macro*⟩`}`

When this handler is installed, each time a plot stream command is called, the same command will be appended to ⟨*macros*⟩. Thus, at the end of the stream, the ⟨*macro*⟩ will contain all the commands that were issued on the stream. You can then install another handler and invoke ⟨*macro*⟩ to "replay" the stream (possibly many times).



```
\begin{pgfpicture}
  \pgfplothandlerrecord{\mystream}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfplothandlerlineto
  \mystream
  \pgfplothandlerclosedcurve
  \mystream
  \pgfusepath{stroke}
\end{pgfpicture}
```

26 Quick Commands

This section explains the "quick" commands of PGF. These commands are executed more quickly than the normal commands of PGF, but offer less functionality. You should use these commands only if you either have a very large amount of commands that need to be processed or if you expect your commands to be executed very often.

26.1 Quick Path Construction Commands

The difference between the quick and the normal path commands is that the quick path commands

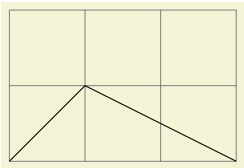
- do not keep track of the bounding boxes,
- do not allow you to arc corners,
- do not apply coordinate transformations.

However, the do use the soft-path subsystem (see Section 29 for details), which allows you to mix quick and normal path commands arbitrarily.

All quick path construction commands start with `\pgfpathq`.

`\pgfpathqmoveto{`⟨*x dimension*⟩`}{`⟨*y dimension*⟩`}`

Either starts a path or starts a new part of a path at the coordinate (⟨*x dimension*⟩, ⟨*y dimension*⟩). The coordinate is *not* transformed by the current coordinate transformation matrix. However, any lowlevel transformations apply.



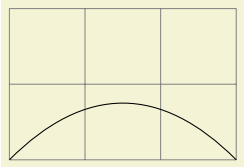
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % no effect
  \pgfpathqlineto{1cm}{1cm} % no effect
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathqlineto{`⟨*x dimension*⟩`}{`⟨*y dimension*⟩`}`

The quick version of the line-to operation.

`\pgfpathqcurveto{`⟨s_x^1⟩`}{`⟨s_y^1⟩`}{`⟨s_x^2⟩`}{`⟨s_y^2⟩`}{`⟨t_x⟩`}{`⟨t_y⟩`}`

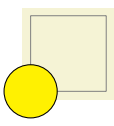
The quick version of the curve-to operation. The first support point is (s_x^1, s_y^1), the second support point is (s_x^2, s_y^2), and the target is (t_x, t_y).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathqmoveto{0pt}{0pt}
  \pgfpathqcurveto{1cm}{1cm}{2cm}{1cm}{3cm}{0cm}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathqcircle{`⟨*radius*⟩`}`

Adds a raduis around the origin of the given ⟨*raduis*⟩. This command is orders of magnitude faster than `\pgfcircle{\pgfpointorigin}{`metaradius`}`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (1,1);
  \pgfpathqcircle{10pt}
  \pgfsetfillcolor{yellow}
  \pgfusepath{stroke,fill}
\end{tikzpicture}
```

26.2 Quick Path Usage Commands

The quick path usage commands perform similar tasks as `\pgfusepath`, but the they

- do not add arrows,
- do not modify the path in any way, in particular,
- ends are not shortened,
- corners are nto replaced by arcs.

Note that you *have to* use the quick versions in the code of arrow definitions since, inside these definition, you obviously do not want arrows to be drawn.

`\pgfusepathqstroke`

Strokes the path without further ado. No arrows are drawn, no corners are arced.

```
\begin{pgfpicture}
  \pgfpathqcircle{5pt}
  \pgfusepathqstroke
\end{pgfpicture}
```

`\pgfusepathqfill`

Fills the path without further ado.

`\pgfusepathqfillstroke`

Fills and then strokes the path without further ado.

`\pgfusepathqclip`

Clips all subsequent drawings against the current path. The path is not processed.

26.3 Quick Text Box Commands

`\pgfqbox{`⟨*box*⟩`}`

This command inserts a TEX box into a `{pgfpicture}` by "escaping" to TEX, inserting the ⟨*box*⟩ at the origin, and then returning to the typesetting the picture.

The ⟨*box*⟩ *must* have a height, width, and depth of zero points. Otherwise, the output may become corrupted.

Part V
The System Layer

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.

27 Design of the System Layer

27.1 Driver Files

The PGF system layer consists of a large number of commands starting with `\pgfsys@`. These commands will be called *system commands* in the following. The higher layers "interface" with the system layer by calling these commands. The higher layers should never use `\special` commands directly or even check whether `\pdfoutput` is defined. Instead, all drawing requests should be "channeled" through the system commands.

The system layer is loaded and setup by the following package:

```
\usepackage{pgfsys} % LaTeX
\input pgfsys.tex   % plain TeX
\input pgfsys.tex   % ConTeX
```

This file provides "default implementations" of all system commands, but most simply produce a warning that they are not implemented. The actual implementations of the system commands for a particular driver like, say, `pdftex` reside in files called `pgfsys-pdftex.sty`. These will be called *driver files* in the following.

When `pgfsys.sty` is loaded, it will try to determine which driver is used by loading `pgf.cfg`. This file should setup the macro `\pgfsysdriver` appropriately. The, `pgfsys.sty` will input the appropriate `pgfsys-`⟨*drivername*⟩`.sty`.

`\pgfsysdriver`

This macro should expand to the name of the driver to be used by `pgfsys`. The default from `pgf.cfg` is `pgfsys-\Gin@driver`. This is very likely to be correct if you are using LaTeX. For plain TeX, the macro will be set to `pgfsys-pdftex.def` if `pdftex` is used und to `pgfsys-dvips.def` otherwise.

File `pgf.cfg`

This file should setup the command `\pgfsysdriver` correctly. If `\pgfsysdriver` is already set to some value, the driver normally should not change it. Otherwise, it should make a "good guess" at which driver will be appropriate.

27.2 System Commands Shared Between Different Drivers

Some definitions of system layer commands can be "shared" between different drivers. For example, the literal text needed to stroke a path in pdf is `S`, independently of the driver. For this reason, the drivers for `pfdtex` and for `dvipdfm`, both of which produce `.pdf` in the end, both include the file `pgfsys-common-pdf.def`, which defines all common commands. Similarly, all PostScript based drivers can used `pgfsys-common-postscript.def` for the "standard" postscript commands.

27.3 Existing Driver Files

With the current version of PGF, the following drivers are implemented:

27.4 Supported Drivers

File `pgfsys-pdftex.def`

This is a driver file for use with pdfTeX, that is, with the `pdftex` or `pdflatex` command. It includes `pgfsys-common-pdf.def`. This driver has the most functionality.

File `pgfsys-dvipdfm.def`

This is a driver file for use with (`la`)`tex` followed by `dvipdfm`. It includes `pgfsys-common-pdf.def`. This driver uses `graphicx` for the graphics inclusion and does not support masking. It does not support image inclusion in plain TeX mode.

File `pgfsys-dvips.def`

This is a driver file for use with (`la`)`tet` followed by `dvips`. It includes `pgfsys-common-postscript.def`. This driver uses `graphicx` for the graphics inclusion and does not support masking. Shading is implemented, but the results will not be as good as with a driver producing `.pdf` as output. It does not support image inclusion in plain TeX mode.

27.5 Common Definition Files

Some drivers share many `\pgfsys@` commands. For the reason, files defining these "common" commands are available. These files are *not* usable alone.

File pgfsys-common-postscript

This file defines `\pgfsys@` commands so that they produce appropriate PostScript code.

File pgfsys-common-pdf

This file defines `\pgfsys@` commands so that they produce appropriate PDF code.

28 Commands of the System Layer

28.1 Beginning and Ending a Stream of System Commands

A "user" of the PGF system layer (like the basic layer or a frontend) will interface with the system layer by calling a stream of commands starting with `\pgfsys@`. From the system layer's point of view, these commands form a long stream. Between calls to the system layer, control goes back to the user.

The driver files implement system layer commands by inserting `\special` commands the implement the desired operation. For example, `\pgfsys@stroke` will be mapped to `\special{pdf: S}` by the driver file for `pdftex`.

For many drivers, when such a stream of specials starts, it is necessary to install an appropriate transformation and perhaps perform some more burocratic tasks. For this reason, every stream will start with a `\pgfsys@beginpicture` and will end with a corresponding ending command.

`\pgfsys@beginpicture`

Called at the beginning of a `{pgfpicture}`. By default, this just opens a scope.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@endpicture`

Called at the end of a pgfpicture. By default, this discards the path and closes the scope.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@beginpurepicture`

This version of the `\pgfsys@beginpicture` picture command can be used for pictures that are guaranteed not to contain any escaped hboxes (see below). In this case, a driver might provide a more compact version of the command.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@endpurepicture`

Called at the end of a "pure" `{pgfpicture}`.

This command has a default implementation and need not be implemented by a driver file.

Inside a stream it is sometimes necessary to "escape" back into normal typesetting mode; for example to insert some normal text, but with all of the current transformations and clippings being in force. For this escaping, the following commands are used:

`\pgfsys@beginhbox`

Called before a TeX hbox is typeset inside a pgfpicture. By default, this just opens a scope.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@endhbox`

Called after a TeX hbox has been typeset inside a `{pgfpicture}`. By default, this discards the path and closes the scope.

This command has a default implementation and need not be implemented by a driver file.

28.2 Path Construction System Commands

`\pgfsys@moveto{`$\langle x \rangle$`}{`$\langle y \rangle$`}`

This command is used to start a path at a specific point (x, y) or to move the current point of the current path to (x, y) without drawing anything upon stroking (the current path is "interrupted").

Both $\langle x \rangle$ and $\langle y \rangle$ are given as TEX dimensions. It is the driver's job to transform these to the coordinate system of the backend. Typically, this means converting the TEX dimension into a dimensionless multiple of $\frac{1}{72}$in. The function `\pgf@sys@bp` helps with this conversion.

Example: Draw a line from (10pt, 10pt) to the origin of the picture.

```
\pgfsys@moveto{10pt}{10pt}
\pgfsys@lineto{0pt}{0pt}
\pgfsys@stroke
```

This command is protocolled, see Section 30.

\pgfsys@lineto{⟨x⟩}{⟨y⟩}

Continue the current path to (x, y) with a straight line.

This command is protocolled, see Section 30.

\pgfsys@curveto{⟨x1⟩}{⟨y1⟩}{⟨x2⟩}{⟨y2⟩}{⟨x3⟩}{⟨y3⟩}

Continue the current path to (x_3, y_3) with a Beziér curve that has the two control points (x_1, y_1) and (x_2, y_2).

Example: Draw a good approximation of a quarter circle:

```
\pgfsys@moveto{10pt}{0pt}
\pgfsys@curveto{10pt}{5.55pt}{5.55pt}{10pt}{0pt}{10pt}
\pgfsys@stroke
```

This command is protocolled, see Section 30.

\pgfsys@rect{⟨x⟩}{⟨y⟩}{⟨width⟩}{⟨height⟩}

Append a rectangle to the current path whose lower left corner is at (x, y) and whose width and height in big points are given by ⟨*width*⟩ and ⟨*height*⟩.

This command can be "mapped back" to `\pgfsys@moveto` and `\pgfsys@lineto` commands, but it is included since PDF has a special, quick version of this command.

This command is protocolled, see Section 30.

\pgfsys@closepath

Close the current path. This results in joining the current point of the path with the point specified by the last `\pgfsys@moveto` operation. Typically, this is preferable over using `\pgfsys@lineto` to the last point specified by a `\pgfsys@moveto`, since the line starting at this point and the line ending at this point will be smoothly joined by `\pgfsys@closepath`.

Example: Consider

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@closepath
\pgfsys@stroke
```

and

```
\pgfsys@moveto{0bp}{0bp}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@lineto{0bp}{0bp}
\pgfsys@stroke
```

The difference between the above will be that in the second triangle the corner at the origin will be wrong; it will just be the overlay of two lines going in different directions, not a sharp pointed corner.

This command is protocolled, see Section 30.

28.3 Coordinate System Transformation System Commands

\pgfsys@transformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨e⟩}{⟨f⟩}

Perform a concatenation of the canvas transformation matrix with the matrix given by the values ⟨a⟩ to ⟨f⟩, see the PDF or PostScript manual for details. The values ⟨a⟩ to ⟨d⟩ are dimensionless factors, ⟨e⟩ and ⟨f⟩ are TEX dimensions

Example: `\pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}`.

This command is protocolled, see Section 30.

\pgfsys@transformshift{⟨*x displacement*⟩}{⟨*y displacement*⟩}

This command will change the origin of the canvas to (x, y).

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 30.

\pgfsys@transformxyscale{⟨*x scale*⟩}{⟨*y scale*⟩}

This command will scale the canvas (and everything that is drawn) by a factor of ⟨*x scale*⟩ in the x-direction and ⟨*y scale*⟩ in the y-direction. Note that this applies to everything, including lines. So a scaled line will have a different width and may even have a different width when going along the x-axis and when going along the y-axis, if the scaling is different in these directions. Usually, you do not want this.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 30.

28.4 Stroking, Filling, and Clipping System Commands

\pgfsys@stroke

Stroke the current path (as if it were drawn with a pen). A number of graphic state parameters influence this, which can be set using appropriate system commands described later.

linewidth The "thickness" of the line. A width of 0 is the thinnest width renderable on the device. On a high-resolution printer this may become invisible and should be avoided. A good choice is 0.4pt, which is the default.

stroke color This special color is used for stroking. If it is not set, the current color is used.

cap The cap describes how the endings of lines a drawn. A round cap adds a little half circle to these endings. A butt cap ends the lines exactly at the end (or start) point without anything added. A rectangular cap ends the lines like the butt cap, but the lines protrude over the endpoint by the line thickness. (See also the PDF manual). If the path has been closed, no cap is drawn.

join This describes how a bend (a join) in a path is rendered. A round join draws bends using small arcs. A bevel join just draws the two lines and then fills the join minimally so that it becomes convex. A miter join extends the lines so that they form a single sharp corner, but only up to a certain miter limit. (See the PDF manual once more).

dash The line may be dashed according to a dashing pattern.

clipping area If a clipping area is established, only those parts of the path that are inside the clipping area will be drawn.

In addition to stroking a path, the path may also be used for clipping after it has been stroked. This will happen if the `\pgfsys@clipnext` is used prior to this command, see there for details.

This command is protocolled, see Section 30.

\pgfsys@closestroke

This command should have the same effect as first closing the path and then stroking it.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 30.

\pgfsys@fill

This command fills the area surrounded by the current path. If the path has not yet been closed, it is closed prior to filling. The path itself is not stroked. For self-intersecting paths or paths consisting of multiple parts, the nonzero winding number rule is used to determine whether a point is inside or outside the path, except if `\ifpgfsys@eorule` holds – in which case the even-odd rule should be used. (See the PDF or PostScript manual for details.)

The following graphic state parameters influence the filling:

eo rule If `\ifpgfsys@eorule` is set, the even-odd rule is used, otherwise the non-zero winding number rule.

fill If the fill color is not especially set, the current color is used.

clipping area If a clipping area is established, only those parts of the filling area that are inside the clipping area will be drawn.

In addition to filling the path, the path will also be used for clipping if `\pgfsys@clipnext` is used prior to this command.

This command is protocolled, see Section 30.

\pgfsys@fillstroke

First, the path is filled, then the path is stroked. If the fill and stroke colors are the same (or if they are not specified and the current color is used), this yields almost the same as a `\pgfsys@fill`. However, due to the line thickness of the stroked path, the fillstroked area will be slightly larger.

In addition to stroking and filling the path, the path will also be used for clipping if `\pgfsys@clipnext` is used prior to this command.

This command is protocolled, see Section 30.

\pgfsys@discardpath

Normally, this command should 'throw away' the current path. However, after `\pgfsys@clipnext` has been called, the current path should subsequently be used for clipping. See `\pgfsys@clipnext` for details.

This command is protocolled, see Section 30.

\pgfsys@clipnext

This command should be issued after a path has been constructed, but before it has been stroked and/or filled or discarded. When the command is used, the next stroking/filling/discarding command will first be executed normally. Then, afterwards, the just-used path will be used for subsequent clipping. If there has already been a clipping region, this region is intersected with the new clipping path (the clipping cannot get bigger). The nonzero winding number rule is used to determine whether a point is inside or outside the clipping area or the even-odd rule, depending on whether `\ifpgfsys@eorule` holds.

28.5 Graphic State Option System Commands

\pgfsys@setlinewidth{⟨*width*⟩}

Sets the width of lines, when stroked, to ⟨*width*⟩, which must be a TEX dimension.

This command is protocolled, see Section 30.

\pgfsys@buttcap

Sets the cap to a butt cap. See `\pgfsys@stroke`.

This command is protocolled, see Section 30.

\pgfsys@roundcap

Sets the cap to a round cap. See `\pgfsys@stroke`.

This command is protocolled, see Section 30.

\pgfsys@rectcap

Sets the cap to a rectangular cap. See \pgfsys@stroke.

This command is protocolled, see Section 30.

\pgfsys@miterjoin

Sets the join to a miter join. See \pgfsys@stroke.

This command is protocolled, see Section 30.

\pgfsys@setmiterlimit{⟨*dimension*⟩}

Sets the miter limit of lines to ⟨*dimension*⟩ big points. See the PDFor PostScript for details on what the miter limit is.

This command is protocolled, see Section 30.

\pgfsys@roundjoin

Sets the join to a round join. See \pgfsys@stroke.

This command is protocolled, see Section 30.

\pgfsys@beveljoin

Sets the join to a bevel join. See \pgfsys@stroke.

This command is protocolled, see Section 30.

\pgfsys@setdash{⟨*pattern*⟩}{⟨*phase*⟩}

Sets the dashing patter. ⟨*pattern*⟩ should be a list of TEX dimensions lengths separated by commas. ⟨*phase*⟩ should be a single dimension.

Example: \pgfsys@setdash{3pt,3pt}{0pt}

The list of values in ⟨*pattern*⟩ is used to determine the lengths of the 'on' phases of the dashing and of the 'off' phases. For example, if ⟨*pattern*⟩ is '3bp,4bp', then the dashing pattern is "3bp on followed by 4bp off, followed by 3bp on, followed by 4bp off, and so on." A pattern of ".5 4 3 1.5" means ".5bp on, 4bp off, 3bp on, 1.5bp off, .5bp on, ..." If the number of entries is odd, the last one is used twice, so "3" means "3bp on, 3bp off, 3bp on, 3bp off, ..." An empty list means "always on."

The second argument determines the "phase" of the pattern. For example, for a pattern of "3bp,4bp" and a phase of "1bp", the pattern would start: "2bp on, 4bp off, 3bp on, 4bp off, 3bp on, 4bp off, ..."

This command is protocolled, see Section 30.

\ifpgfsys@eorule

Determines whether the even odd rule is used for filling ang clipping or not.

28.6 Color System Commands

The PGF system layer provides a number of system commands for setting colors. These command coexist with commands from the color and xcolor package, which perform similar functions. However, the color package does not support having two different colors for stroking and filling, which is a useful feature that is supported by PGF. For this reason, the PGF system layer offers commands for setting these colors separatedly. Also, plain TEX profits from the fact that PGF can set colors.

For PDF, implementing these color commands is easy since PDF supports different stroking and filling colors directly. For PostScript, a more complicated approach is needed in which the colors need to be stored in special PostScript variables that are set wheneven a stroking or a filling operation is done.

\pgfsys@color@rgb@{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for stroking and filling operations to the given red/green/blue tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@rgb@stroke{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for stroking operations to the given red/green/blue tuple (numbers between 0 and 1).

Example: Make stroked text dark red: `\pgfsys@color@rgb@stroke{0.5}{0}{0}`

The special stroking color is only used if the stroking color has been set since the last `\color` or `\pgfsys@color@xxx` command. Thus, each `\color` command will reset both the stroking and filling colors by calling `\pgfsys@color@reset`.

This command is protocolled, see Section 30.

\pgfsys@color@rgb@fill{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for filling operations to the given red/green/blue tuple (numbers between 0 and 1). This color may be different from the stroking color.

This command is protocolled, see Section 30.

\pgfsys@color@cmyk{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for stroking and filling operations to the given cymk tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@cmyk@stroke{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for stroking operations to the given cymk tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@cmyk@fill{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for filling operations to the given cymk tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@cmy{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for stroking and filling operations to the given cym tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@cmy@stroke{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for stroking operations to the given cym tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@cmy@fill{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for filling operations to the given cym tuple (numbers between 0 and 1).

This command is protocolled, see Section 30.

\pgfsys@color@gray{⟨*black*⟩}

Sets the color used for stroking and filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 30.

\pgfsys@color@gray@stroke{⟨*black*⟩}

Sets the color used for stroking operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 30.

\pgfsys@color@gray@fill{⟨*black*⟩}

Sets the color used for filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 30.

\pgfsys@color@reset

This command will be called when the \color command is used. It should purge any internal settings of stroking and filling color. After this call, till the next use of a command like \pgfsys@color@rgb@fill, the current color installed by the \color command should be used.

If the TEX-if \pgfsys@color@reset@inorder is set to true, this command may "assume" that any call to a color command that sets the fill or stroke color came "before" the call to this command and may try to optimize the output accordingly.

An example of an incorrect "out of order" call would be using \pgfsys@color@reset at the beginning of a bos that is constructed using \setbox. Then, when the box is constructed, no special fill or stroke color might be in force. However, when the box is later on inserted at some point, a special fill color might already have been set. In this case, this command is not guaranteed to reset the color correctly. Use \pgfsys@color@reset@outoforder in such cases.

\pgfsys@color@reset@inordertrue

Sets the optimized "in order" version of the color resetting. This is the default.

\pgfsys@color@reset@inorderfalse

Switches off the optimized color resetting.

\pgfsys@color@unstacked{⟨LaTeX color⟩}

This slightly obscure command causes the color stack to be tricked. When called, this command should set the current color to ⟨LaTeX color⟩ without causing any change in the color stack.

Example: \pgfsys@color@unstacked{red}

28.7 Scoping System Commands

The scoping commands are used to keep changes of the graphics state local.

\pgfsys@beginscope

Saves the current graphic state on a graphic state stack. All changes to the graphic state parameters mentioned for \pgfsys@stroke and \pgfsys@fill will be local to the current graphic state and will the old values will be restored after endscope is used.

Warning: PDFand PostScript differ with respect to the question of whether the current path is part of the graphic state or not. For this reason, you should never use this command unless the path is currently empty. For example, it might be a good idea to use discardpath prior to calling this command.

This command is protocolled, see Section 30.

\pgfsys@endscope

Restores the last saved graphic state.

This command is protocolled, see Section 30.

28.8 Image System Commands

The sytem layer provides some commands for image inclusion. However, this whole subsystem is not really well-designed and the graphics package does a better job at image inclusion than PGF does. However, this subsystem is currently still needed since only PGF supports image masking. Once this feature is added to the graphics package, the whole subsystem will be mapped back to graphics and become obsolet.

\pgfsys@imagesuffixlist

This macro should expand to a list of suffixes, separated by ':', that will be tried when searching for an image.

Example: \def\pgfsys@imagesuffixlist{eps:epsi:ps}

\pgfsys@defineimage

Called, when an image should be defined.

This command does not take any parameters. Instead, certain macros will be preinstalled with appropriate values when this command is invoked. These are:

- \pgf@filename File name of the image to be defined.
- \pgf@imagewidth Will be set to the desired (scaled) width of the image.
- \pgf@imageheight Will be set to the desired (scaled) height of the image.

  If this macro and also the height macro are empty, the image should have its 'natural' size.

  If exactly only of them is specified, the undefined value the image is scaled so that the aspect ratio is kept.

  If both are set, the image is scaled in both directions independently, possibly changing the aspect ratio.

The following macros presumable mostly make sense for drivers that can handle PDF:

- \pgf@imagepage The desired page number to be extracted from a multi-page "image."
- \pgf@imagemask If set, it will be set to `/SMask x 0 R` where `x` is the pdf object number of a soft mask to be applied to the image.
- \pgf@imageinterpolate If set, it will be set to `/Interpolate true` or `/Interpolate false`, indicating whether the image should be interpolated in PDF.

The command should now setup the macro \pgf@image such that calling this macro will result in typesetting the image. Thus, \pgf@image is the "return value" of the command.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@definemask

This command declares a mask for usage with images. It works similar to \pgfsys@defineimage: Certain macros are set when the command is called. The result should be to set the macro \pgf@mask to a pdf object count that can subsequently be used as a soft mask. The following macros will be set when this command is invoked:

- \pgf@filename File name of the mask to be defined.
- \pgf@maskmatte The so-called matte of the mask (see the PDF documentation for details). The matte is a color specification consisting of 1, 3 or 4 numbers between 0 and 1. The number of numbers depends on the number of color channels in the image (not in the mask!). It will be assumed that the image has been preblended with this color.

28.9 Shading System Commands

\pgfsys@horishading{⟨*name*⟩}{⟨*height*⟩}{⟨*specification*⟩}

Declares a horizontal shading for later use. The effect of this command should be the definition of a macro called `@pgfshading`metaname`!` (or `\csname @pdfshading`⟨*name*⟩`!\endcsname`, to be precise). When invoked, this new macro should insert a shading at the current position.

⟨*name*⟩ is the name of the shading, which is also used in the output macro name. ⟨*height*⟩ is the height of the shading and must be given as a TeX dimension like `2cm` or `10pt`. ⟨*specification*⟩ is a shading color specification as specified in Section 23. The shading specification implicitly fixes the width of the shading.

When `@pgfshading`metaname`!` is invoked, it should insert a box of height ⟨*height*⟩ and the width implicit in the shading declaration.

\pgfsys@vertshading{⟨*name*⟩}{⟨*width*⟩}{⟨*specification*⟩}

Like the horizontal version, only for vertical shadings. This time, the height of the shading is implicit in ⟨*specification*⟩ and the width is given as ⟨*width*⟩.

\pgfsys@radialshading{⟨*name*⟩}{⟨*starting point*⟩}{⟨*specification*⟩}

Declares a radial shading. Like the previous macros, this command should setup the macro `@pgfshading`metaname`!`, which upon invocation should insert a radial shading whose size is implicit in ⟨*specification*⟩.

The parameter ⟨*starting point*⟩ is a pgf point specification if the starting point of the shading.

28.10 Reusable Objects System Commands

\pgfsys@invoke{⟨*literals*⟩}

This command gets protocolled literals and should insert them into the `.pdf` or `.dvi` file using an appropriate `\special`.

\pgfsys@defobject{⟨*name*⟩}{⟨*lower left*⟩}{⟨*upper right*⟩}{⟨*code*⟩}

Declares an object for later use. The idea is that the object can be precached in some way and then be rendered more quickly when used several times. For example, an arrow head might be defined and prerendered in this way.

The parameter ⟨*name*⟩ is the name for later use. ⟨*lower left*⟩ and ⟨*upper right*⟩ are PGF points specifying a bounding box for the object. ⟨*code*⟩ is the code for the object. The code should not be too fancy.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@useobject{⟨*name*⟩}{⟨*extra code*⟩}

Renders a previously declared object. The first parameter is the name of the the object. The second parameter is extra code that should be executed right *before* the object is rendered. Typically, this will be some transformation code.

This command has a default implementation and need not be implemented by a driver file.

28.11 Invisibility System Commands

All drawing or stroking or text rendering between calls of the following commands should be suppressed. A similar effect can be achieved by clipping against an empty region, but the following commands do not open a graphics scope and can be opened and closed "orthogonally" to other scopings.

\pgfsys@begininvisible

Between this command and the closing endinvisible, all output should be suppressed. Nothing should be drawn at all, which includes all paths, images and shadings. However, no groups (neither TEX groups nor graphic state groups) should be opened by this command.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 30.

\pgfsys@endinvisible

Ends the invisibilty section, unless invisibility blocks have been nested. In this case, only the "last" one restores visibility.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 30.

28.12 Internal Conversion Commands

The system commands take TEX dimensions as input, but the dimensions that have to be inserted into PDF and PostScript files need to be dimensionless values that are interpreted as multiples of $\frac{1}{72}$in. For example, the TEX dimension $2bp$ should be inserted as `2` into a PDF file and the TEX dimension $10pt$ as `9.9626401`. To make this conversion easier, the following command may be useful:

\pgf@sys@bp{⟨*dimensions*⟩}

Inserts how many multiples of $\frac{1}{72}$in the ⟨*dimension*⟩ is into the current protocol stream (buffered).

Example: `\pgf@sys@bp{\pgf@x}` or `\pgf@sys@bp{1cm}`.

Note that this command is *not* a system command that can/needs to be overwritten by a driver.

29 The Soft Path Subsystem

This section describes a set of commands for creating *soft paths* as opposed to the commands of the previous section, which created *hard paths*. A soft path is a path that can still be "changed" or "molded." Once you (or the PGF system) is satisfied with a soft path, it is turned into a hard path, which can be inserted into the resulting `.pdf` or `.ps` file.

Note that the commands described in this section are "high-level" in the sense that they are not implemented in driver files, but rather directly by the PGF-system layer. For this reason, the commands for creating soft paths do not start with `\pgfsys@`, but rather with `\pgfsyssoftpath@`. On the other hand, as a user you will never use these commands directly, so they are described as part of the low-level interface.

29.1 Path Creating Process

When the user writes a command like `\draw (0bp,0bp) -- (10bp,0bp);` quite a lot happens behind the scenes:

1. The frontend command is translated by `tikz` into commands of the basic layer. In essence, the command is translated to something like

   ```
   \pgfpathmoveto{\pgfpoint{0bp}{0bp}}
   \pgfpathlineto{\pgfpoint{10bp}{0bp}}
   \pgfusepath{stroke}
   ```

2. The `\pgfpathxxxx` command do *not* directly call "hard" commands like `\pgfsys@xxxx`. Instead, the command `\pgfpathmoveto` invokes a special command called `\pgfsyssoftpath@moveto` and `\pgfpathlineto` invokes `\pgfsyssoftpath@lineto`.

   The `\pgfsyssoftpath@xxxx` commands, which are described below, construct a soft path. Each time such a command is used, special tokens are added to the end of an internal macro that stores the soft path currently being constructed.

3. When the `\pgfusepath` is encountered, the soft path stored in the internal macro is "invoked." Only now does a special macro iterate over the soft path. For each lineto or moveto operation on this path it calls an appropriate `\pgfsys@moveto` or `\pgfsys@lineto` in order to, finally, create the desired hard path, namely, the string of literals in the `.pdf` or `.ps` file.

4. After the path has been invoked, `\pgfsys@stroke` is called to insert the literal for stroking the path.

Why such a complicated process? Why not have `\pgfpathlineto` directly call `\pgfsys@lineto` and be done with it? There are two reasons:

1. The PDF specification requires that a path is not interrupted with any non-path-construction commands. Thus, the following code will result in a corrupted `.pdf`:

   ```
   \pgfsys@moveto{0}{0}
   \pgfsys@setlinewidth{1}
   \pgfsys@lineto{10}{0}
   \pgfsys@stroke
   ```

   Such corrupt code is *tolerated* by most viewers, but not always. It is much better to create only (reasonably) legal code.

2. A soft path can still be changed, while a hard path is fixed. For example, one can still change the starting and end points of a soft path or do optimizations it. Such transformations are not possible on hard paths.

29.2 Starting and Ending a Soft Path

No special action must be taken in order to start the creation of a soft path. Rather, each time a command like `\pgfsyssoftpath@lineto` is called, a special token is added to the (global) current soft path being constructed.

However, you can access and change the current soft path. In this way, it is possible to store a soft path, to manipulate it, or to invoke it.

\pgfsyssoftpath@getcurrentpath{⟨*macro name*⟩}

This command will store the current soft path in ⟨*macro name*⟩.

\pgfsyssoftpath@setcurrentpath{⟨*macro name*⟩}

This command will set the current soft path to be the path stored in ⟨*macro name*⟩. This macro should store a path that has previously been extracted using the `getcurrentpath` command and has possibly been modified subsequently.

\pgfsyssoftpath@invokecurrentpath

This command will turnn the current soft path in a "hard" path. To do so, it iterates over the soft path and calls an appropriate `\pgfsys@xxxx` command for each element of the path. Note that the current soft path is *not changed* by this command. Thus, in order to start a new soft path after the old one has been invoked and is no longer needed, you need to set the current soft path to be empty. This may seems strange, but it is often useful to immediately use the last soft path again.

\pgfsyssoftpath@flushcurrentpath

This command will invoke the current soft path and then set it to be empty.

29.3 Soft Path Creation Commands

\pgfsyssoftpath@moveto{⟨x⟩}{⟨y⟩}

This command appends a "moveto" segment to the current soft path. The coordinates ⟨x⟩ and ⟨y⟩ are given as big points, as always in the system level.

Example: One way to draw a line:

```
\pgfsyssoftpath@moveto{0}{0}
\pgfsyssoftpath@lineto{10}{10}
\pgfsyssoftpath@flushcurrentpath
\pgfsys@stroke
```

\pgfsyssoftpath@lineto{⟨x⟩}{⟨y⟩}

Appends a "lineto" segment to the current soft path.

\pgfsyssoftpath@curevto{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨x⟩}{⟨y⟩}

Appends a "curveto" segment to the current soft path with controls (a, b) and (c, d).

\pgfsyssoftpath@rect{⟨*lower left x*⟩}{⟨*lower left y*⟩}⟨*width*⟩⟨*height*⟩

Appends a rectangle segment to the current soft path.

\pgfsyssoftpath@closepath

Appends a "closepath" segment to the current soft path.

29.4 The Soft Path Data Structure

A soft path is stored in a standarized way, which makes it possible to modify it before it becomes "hard." Basically, a soft path is a long sequence of triples. Each triple starts with a *token* that identifies what is going on. This token is followed by two numbers in braces. For example, the following is a soft path that means "the path starts at (0 bp, 0 bp) and then continues in a straight line to (10 bp, 0 bp)."

```
\pgfsyssoftpath@movetotoken{0}{0}\pgfsyssoftpath@linetotoken{10}{0}
```

A curveto is hard to express in this way since we need six numbers to express it, not two. For this reasons, a curveto is expressed using three triples as follows: A `\pgfsyssoftpath@curevto{1}{2}{3}{4}{5}{6}` results in the following three triples:

```
\pgfsyssoftpath@curvetosupportatoken{1}{2}
\pgfsyssoftpath@curvetosupportbtoken{3}{4}
\pgfsyssoftpath@curvetotoken{5}{6}
```

These three triples must always "remain together." Thus, a lonely `supportbtoken` is forbidden.
In details, the following tokens exist:

- \pgfsyssoftpath@movetotoken indicates a moveto operation. The two following numbers indicate the position to which the current point should be moved.
- \pgfsyssoftpath@linetotoken indicates a lineto operation.
- \pgfsyssoftpath@curvetosupportatoken indicates the first control point of a curveto operation. The triple must be followed by a \pgfsyssoftpath@curvetosupportbtoken.
- \pgfsyssoftpath@curvetosupportbtoken indicates the second control point of a curveto operation. The triple must be followed by a \pgfsyssoftpath@curvetotoken.
- \pgfsyssoftpath@curvetotoken indicates the target of a curveto operation.
- \pgfsyssoftpath@rectcornertoken indicates the corner of a rectangle on the soft path. The triple must be followed by a \pgfsyssoftpath@rectsizetoken.
- \pgfsyssoftpath@rectcornertoken indicates the size of a rectangle on the soft path.
- \pgfsyssoftpath@closepath indicates that the subpath begun with the last moveto operation should be closed. The parameter numbers are currently not important, but if set to anything different from {0}{0}, they should be set to the coordinate of the original moveto operation to which the path "returns" now.

30 The Protocol Subsystem

This section describes commands for *protocolling* literal text created by PGF. The idea is that some literal text, like the string of commands used to draw an arrow head, will be used over and over again in a picture. It is then much more efficient to compute the necessary literal text just once and to quickly insert it "in a single sweep."

When protocolling is "switched on," there is a "current protocol" to which literal text gets appended. Once all commands that needed to be protocoled have been issued, the protocol can be obtained and stored using `\pgfsysprotocol@getcurrentprotocol`. At any point, the current protocol can be changed using a corresponding setting command. Finally, `\pgfsysprotocol@invokecurrentprotocol` is used to insert the protocoled commands into the `.pdf` or `.dvi` file.

Only those `\pgfsys@` commands can be protocolled that use the command `\pgfsysprotocol@literal` interally. For example, the definition of `\pgfsys@moveto` in `pgfsys-common-pdf.def` is

```
\def\pgfsys@moveto#1#2{\pgfsysprotocol@literal{#1 #2 m}}
```

All "normal" system-level commands can be protocolled. However, commands for creating or invoking shadings, images, or whole pictures require special `\special`'s and cannot be protocolled.

`\pgfsysprotocol@literalbuffered{`⟨*literal text*⟩`}`

Adds the ⟨*literal text*⟩ to the current protocol, after it has been "`\edef`ed." This command will always protocol.

`\pgfsysprotocol@literal{`⟨*literal text*⟩`}`

First calls `\pgfsysprotocol@literalbuffered` on ⟨*literal text*⟩. Then, if protocolling is currently switched off, the ⟨*literal text*⟩ is passed on to `\pgfsys@invoke`.

`\pgfsysprotocol@bufferedtrue`

Turns protocolling on. All subsequent calls of `\pgfsysprotocol@literal` will append their argument to the current protocol.

`\pgfsysprotocol@bufferedfalse`

Turns protocolling off. Subsequent calls of `\pgfsysprotocol@literal` directly insert their argument into the current `.pdf` or `.ps`.

Note that if the current protocol is not empty when protocolling is switched off, the next call to `\pgfsysprotocol@literal` will first flush the current protocol, that is, insert it into the file.

`\pgfsysprotocol@getcurrentprotocol{`⟨*macro name*⟩`}`

Stores the current protocol in ⟨*macro name*⟩ for later use.

`\pgfsysprotocol@setcurrentprotocol{`⟨*macro name*⟩`}`

Sets the current protocol to ⟨*macro name*⟩.

`\pgfsysprotocol@invokecurrentprotocol`

Inserts the text stored in the current protocol into the `.pdf` or `.dvi` file. This does *not* change the current protocol.

`\pgfsysprotocol@flushcurrentprotocol`

First inserts the current protocol, then sets the current protocol to the empty string.

Index

This index only contains automatically generated entries. A good index should also contain carefully selected keywords. This index is not a good index.