XY-pic Reference Manual

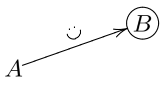
Kristoffer H. Rose ⟨krisrose@brics.dk⟩[×]   Ross Moore ⟨ross@mpce.mq.edu.au⟩[†]

Version 3.7 ⟨1999/02/16⟩

Abstract

This document summarises the capabilities of the XY-pic package for typesetting graphs and diagrams in TEX. For a general introduction as well as availability information and conditions refer to the User's Guide [14].

A characteristic of XY-pic is that it is built around a *kernel drawing language* which is a concise notation for general graphics, *e.g.*,

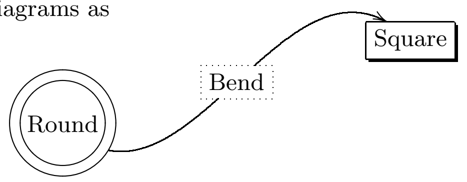


was drawn by the XY-pic kernel code

```
\xy (3,0)*{A} ; (20,6)*+{B}*\cir{} **\dir{-}
    ? *_!/3pt/\dir{)} *_!/7pt/\dir{:}
    ?>* \dir{>} \endxy
```

It is an object-oriented graphic language in the most literal sense: 'objects' in the picture have 'methods' describing how they typeset, stretch, etc. However, the syntax is rather terse.

Particular applications make use of *extensions* that enhance the graphic capabilities of the kernel to handle such diagrams as



which was typeset by

```
\xy *[o]=<40pt>\hbox{Round}="o"*\frm{oo},
    +<5em,-5em>@+,
    (46,11)*+\hbox{Square}="s"  *\frm{-,},
    -<5em,-5em>@+,
 "o";"s" **{} ?*+\hbox{Bend}="b"*\frm{.},
 "o";"s"."b" **\crvs{-},
 "o"."b";"s" **\crvs{-} ?>*\dir{>}
\endxy
```

using the 'curve' and 'frame' extensions.

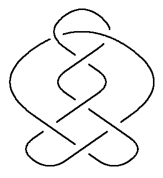
All this is made accessible through the use of *features* that provide convenient notation such that users can enter special classes of diagrams in an intuitive form, *e.g.*, the diagram

$$\begin{tikzcd}
U \arrow[ddr, bend right, "y"'] \arrow[dr] \arrow[drr, bend left, "x"] & & \\
& X \times_Z Y \arrow[r, "p"'] \arrow[d, "q"] & X \arrow[d, "f"'] \\
& Y \arrow[r, "g"] & Z
\end{tikzcd}$$

was typeset using the 'matrix' features by the XY-pic input lines

```
\xymatrix{
 U \ar@/_/[ddr]_y \ar[dr] \ar@/^/[drr]^x \\
  & X \times_Z Y \ar[d]^q \ar[r]_p
               & X \ar[d]_f           \\
  & Y \ar[r]^g   & Z                    }
```

Features exist for many kinds of input; here is a knot typeset using the 'knots and links' feature:



The current implementation is programmed entirely within "standard TEX and METAFONT", *i.e.*, using TEX macros (no `\specials`) and with fonts designed using METAFONT. Optionally special 'drivers' make it possible to produce DVI files with 'specials' for extra graphics capabilities, *e.g.*, using POSTSCRIPT.[1]

[×]BRICS, Computer Science, Aarhus University (bld. 540), Ny Munkegade, DK–8000 Aarhus C, Denmark.
[†]MPCE (Mathematics dept.), Macquarie University, North Ryde, Sydney, Australia NSW 2109.
[1]POSTSCRIPT is a registered Trademark of Adobe, Inc. [1].

Contents

I The Kernel 4

1 The XY-pic implementation 4
- 1.1 Loading XY-pic . . . 4
- 1.2 Logo, version, and messages . . . 5
- 1.3 Fonts . . . 5
- 1.4 Allocations . . . 5

2 Picture basics 6
- 2.1 Positions . . . 6
- 2.2 Objects . . . 6
- 2.3 Connections . . . 6
- 2.4 Decorations . . . 6
- 2.5 The XY-pic state . . . 6

3 Positions 7

4 Objects 11

5 Decorations 15

6 Kernel object library 16
- 6.1 Directionals . . . 16
- 6.2 Circle segments . . . 18
- 6.3 Text . . . 18

7 XY-pic options 18
- 7.1 Loading . . . 19
- 7.2 Option file format . . . 19
- 7.3 Driver options . . . 20

II Extensions 20

8 Curve and Spline extension 20
- 8.1 Curved connections . . . 20
- 8.2 Circles and Ellipses . . . 24
- 8.3 Quadratic Splines . . . 24

9 Frame and Bracket extension 24
- 9.1 Frames . . . 24
- 9.2 Brackets . . . 26
- 9.3 Filled regions . . . 26
- 9.4 Framing as object modifier . . . 27
- 9.5 Using curves for frames . . . 27

10 More Tips extension 27

11 Line styles extension 27

12 Rotate and Scale extension 29

13 Colour extension 30

14 Pattern and Tile extension 31

15 Import graphics extension 33

16 Movie Storyboard extension 34

17 PostScript backend 34
- 17.1 Choosing the DVI-driver . . . 34
- 17.2 Why use PostScript . . . 35

18 TPIC backend 36

19 em-TeX backend 37

20 Necula's extensions 37
- 20.1 Expansion . . . 37
- 20.2 Polygon shapes . . . 37

III Features 37

21 All features 37

22 Dummy option 37

23 Arrow and Path feature 38
- 23.1 Paths . . . 38
- 23.2 Arrows . . . 40

24 Two-cell feature 43
- 24.1 Typesetting 2-cells in Diagrams . . . 43
- 24.2 Standard Options . . . 44
- 24.3 Nudging . . . 44
- 24.4 Extra Options . . . 46
- 24.5 2-cells in general XY-pictures . . . 47

25 Matrix feature 48
- 25.1 XY-matrices . . . 48
- 25.2 New coordinate formats . . . 49
- 25.3 Spacing and rotation . . . 49
- 25.4 Entries . . . 50

26 Graph feature 51

27 Polygon feature 53

28 Lattice and web feature 56

29 Circle, Ellipse, Arc feature 58
- 29.1 Full Circles . . . 58
- 29.2 Ellipses . . . 58
- 29.3 Circular and Elliptical Arcs . . . 59

30 Knots and Links feature 61

31 Smart Path option 66

IV Drivers 66

32 Support for Specific Drivers **66**
32.1 dvidrv driver 66
32.2 DVIPS driver 66
32.3 DVITOPS driver 67
32.4 OzTeX driver 67
32.5 OzTeX v1.7 driver 67
32.6 Textures driver 68
32.7 Textures v1.6 driver 68
32.8 XDVI driver 68
32.9 CMacTeX driver 69

33 Extra features using PostScript drivers **69**
33.1 Colour 70
33.2 Frames 70
33.3 Line-styles 70
33.4 Rotations and scaling 70
33.5 Patterns and tiles 71

34 Extra features using tpic drivers **71**
34.1 frames. 71

Appendices **71**

A Answers to all exercises **71**

B Version 2 Compatibility **75**
B.1 Unsupported incompatibilities 75
B.2 Obsolete kernel features 75
B.3 Obsolete extensions & features 76
B.4 Obsolete loading 77
B.5 Compiling v2-diagrams 77

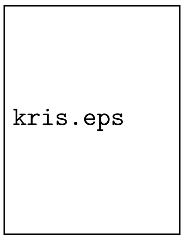
C Common Errors **77**

References **77**

Index **78**

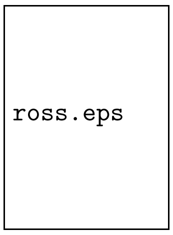
List of Figures

1 ⟨pos⟩itions. 8
2 Example ⟨place⟩s 10
3 ⟨object⟩s. 12
4 ⟨decor⟩ations. 16
5 Kernel library ⟨dir⟩ectionals 17
6 ⟨cir⟩cles. 19
7 Syntax for curves. 22
8 Plain ⟨frame⟩s. 25
9 Bracket ⟨frame⟩s. 25
10 Rotations, scalings and flips 31
11 The 38 standard Macintosh patterns. . 32
12 importing a graphic for labelling . . . 34
13 ⟨path⟩s 39
14 ⟨arrow⟩s. 41
15 Pasting diagram. 44
16 ⟨twocell⟩s 45
17 ⟨graph⟩s 52
18 ⟨knot-piece⟩ construction set 62
19 knot crossings with orientations and label positions 63
20 knot joins, with orientations, labels and shifts 65
21 Extension implementation replaced by use of ⟨driver⟩ specials. 69



Kristoffer Rose

Ross Moore



Preface

This reference manual gives concise descriptions of the modules of XY-pic, written by the individual authors. Please direct any TeXnical question or suggestion for improvement directly to the author of the component in question, preferably by electronic mail using the indicated address. Complete documents and printed technical documentation or software is most useful.

The first part documents the XY-pic kernel which is always loaded. The remaining parts describe the three kinds of options: *extensions* in part II extend the kernel graphic capabilities, *features* in part III provide special input syntax for particular diagram types, and *drivers* in part IV make it possible to exploit the printing capabilities supported by DVI driver programs. For each option it is indicated how it should be loaded. The appendices contain answers to all the exercises, a summary of the compatibility with version 2, and list some reasons why XY-pic might sometimes halt with a cryptic TeX error.

even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this package; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

In practice this means that you are free to use XY-pic for your documents but if you distribute any part of XY-pic (including modified versions) to someone then you are obliged to ensure that the full source text of XY-pic is available to them (the full text of the license in the file `COPYING` explains this in somewhat more detail ☺).

Notational conventions. We give descriptions of the *syntax* of pictures as BNF[2] rules; in explanations we will use upper case letters like X and Y for ⟨dimen⟩sions and lower case like x and y for ⟨factor⟩s.

Part I
The Kernel

Vers. 3.7 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩

After giving an overview of the XY-pic environment in §1, this part document the basic concepts of XY-picture construction in §2, including the maintained 'graphic state'. The following sections give the precise syntax rules of the main XY-pic constructions: the position language in §3, the object constructions in §4, and the picture 'decorations' in §5. §6 presents the kernel repertoire of objects for use in pictures; §7 documents the interface to XY-pic options like the standard 'feature' and 'extension' options.

Details of the implementation are not discussed here but in the complete TEXnical documentation [15].

1 The XY-pic implementation

This section briefly discusses the various aspects of the present XY-pic kernel implementation of which the user should be aware.

1.1 Loading XY-pic

XY-pic is careful to set up its own environment in order to function with a large variety of formats. For most formats a single line with the command

```
\input xy
```

in the preamble of a document file should load the kernel (see 'integration with standard formats' below for variations possible with certain formats, in particular LATEX [9]).

The rest of this section describes things you must consider if you need to use XY-pic together with other macro packages, style options, or formats. The less your environment deviates from plain TEX the easier it should be. Consult the TEXnical documentation [15] for the exact requirements for other definitions to coexist with XY-pic.

Privacy: XY-pic will warn about control sequences it redefines—thus you can be sure that there are no conflicts between XY-pic-defined control sequences, those of your format, and other macros, provided you load XY-pic last and get no warning messages like

```
Xy-pic Warning: '...' redefined.
```

In general the XY-pic kernel will check all control sequences it redefines *except* that (1) generic temporaries like `\next` are not checked, (2) predefined font identifiers (see §1.3) are assumed intentionally preloaded, and (3) some of the more exotic control sequence names used internally (like `@{-}`) are only checked to be different from `\relax`.

Category codes: The situation is complicated by the flexibility of TEX's input format. The culprit is the 'category code' concept of TEX (*cf.* [6, p.37]): when loaded XY-pic requires the characters `␣\{}%` (the first is a space) to have their standard meaning and all other printable characters to have the *same category as when XY-pic will be used*—in particular this means that (1) you should surround the loading of XY-pic with `\makeatother` ... `\makeatletter` when loading it from within a LATEX package, and that (2) XY-pic should be loaded after files that change category codes like the `german.sty` that makes `"` active. Some styles require that you reset the catcodes for every diagram, *e.g.*, with `french.sty` you should use the command `\english` before every `\xymatrix`.
However, it is possible to 'repair' the problem in case any of the characters `#$&'+-.<=>`` ` `` change category code:

```
\xyresetcatcodes
```

will load the file `xyrecat.tex` (version 3.3) to do it.

[2] BNF is the notation for "meta-linguistic formulae" first used by [10] to describe the syntax of the Algol programming language. We use it with the conventions of the TEXbook [6]: '$\longrightarrow$' is read "is defined to be", ' | ' is read "or", and '⟨empty⟩' denotes "nothing"; furthermore, '⟨id⟩' denotes anything that expands into a sequence of TEX character tokens, '⟨dimen⟩' and '⟨factor⟩' denote decimal numbers with, respective without, a dimension unit (like `pt` and `mm`), ⟨number⟩ denotes possibly signed integers, and ⟨text⟩ denotes TEX text to be typeset in the appropriate mode. We have chosen to annotate the syntax with brief explanations of the 'action' associated with each rule; here '$\longleftarrow$' should be read 'is copied from'.

Integration with standard formats This is handled by the `xyidioms.tex` file and the integration as a LaTeX [9] package by `xy.sty`.

xyidioms.tex: This included file provides some common idioms whose definition depends on the used format such that XY-pic can use predefined dimension registers etc. and yet still be independent of the format under which it is used. The current version (3.4) handles plain TeX (version 2 and 3 [6]), AMS-TeX (version 2.0 and 2.1 [16]), LaTeX (version 2.09 [8] and 2ε [9]), AMS-LaTeX (version 1.0, 1.1 [2], and 1.2), and eplain (version 2.6 [3])[3].

xy.sty: If you use LaTeX then this file makes it possible to load XY-pic as a 'package' using the LaTeX 2ε [9] `\usepackage` command:

```
\usepackage [⟨option⟩,...] {xy}
```

where the ⟨option⟩s will be interpreted as if passed to `\xyoption` (*cf.* §7).

The only exceptions to this are the options having the same names as those driver package options of part IV, which appear in *cf.* [4, table 11.2, p.317] or the LaTeX 2ε `graphics` bundle. These will automatically invoke any backend extension required to best emulate the LaTeX 2ε behaviour. (This means that, *e.g.*, `[dvips]` and `[textures]` can be used as options to the `\documentclass` command, with the normal effect.)

The file also works as a LaTeX 2.09 [8] 'style option' although you will then have to load options with the `\xyoption` mechanism described in §7.

1.2 Logo, version, and messages

Loading XY-pic prints a banner containing the version and author of the kernel; small progress messages are printed when each major division of the kernel has been loaded. Any options loaded will announce themself in a similar fashion.

If you refer to XY-pic in your written text (please do ☺) then you can use the command `\Xy-pic` to typeset the "XY-pic" logo. The version of the kernel is typeset by `\xyversion` and the release date by `\xydate` (as found in the banner). By the way, the XY-pic *name*[4] originates from the fact that the first version was little more than support for (x, y) coordinates in a configurable coordinate system where the main idea was that *all* operations could be specified in a manner independent of the orientation of the coordinates. This property has been maintained except that now the package allows explicit absolute orientation as well.

Messages that start with "`Xy-pic Warning`" are indications that something needs your attention; an "`Xy-pic Error`" will stop TeX because XY-pic does not know how to proceed.

1.3 Fonts

The XY-pic kernel implementation makes its drawings using five specially designed fonts:

Font	Characters	Default
`\xydashfont`	dashes	`xydash10`
`\xyatipfont`	arrow tips, upper half	`xyatip10`
`\xybtipfont`	arrow tips, lower half	`xybtip10`
`\xybsqlfont`	quarter circles for hooks and squiggles	`xybsql10`
`\xycircfont`	1/8 circle segments	`xycirc10`

The first four contain variations of characters in a large number of directions, the last contains 1/8 circle segments.

Note: The default fonts are not part of the XY-pic kernel *specification*: they just set a standard for what drawing capabilities should at least be required by an XY-pic implementation. Implementations exploiting capabilitites of particular output devices are in use. Hence the fonts are only loaded by XY-pic if the control sequence names are undefined—this is used to preload them at different sizes or prevent them from being loaded at all.

1.4 Allocations

One final thing that you must be aware of is that XY-pic allocates a significant number of dimension registers and some counters, token registers, and box registers, in order to represent the state and do computations. The current kernel allocates 4 counters, 28 dimensions, 2 box registers,4 token registers, 1 read channel, and 1 write channel (when running under LaTeX; some other formats use slightly more because standard generic temporaries are used). Options may allocate further registers (currently loading *everything* loads 6 dimen-, 3 toks-, 1 box-, and 9 count-registers in addition to the kernel ones).

[3]The 'v2' feature introduces some name conflicts, in order to maintain compatibility with earlier versions of XY-pic.

[4]No description of a TeX program is complete without an explanation of its name.

2 Picture basics

The basic concepts involved when constructing XY-pictures are positions and objects, and how they combine to form the state used by the graphic engine.

The general structure of an XY-picture is as follows:

\xy ⟨pos⟩ ⟨decor⟩ \endxy

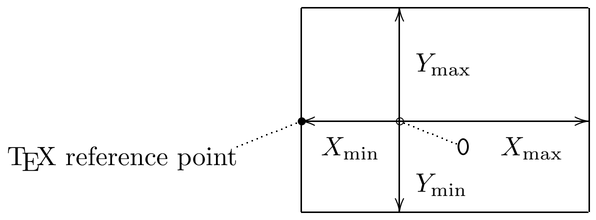
---

builds a box with an XY-picture (LaTeX users may substitute \begin{xy} . . . \end{xy} if they prefer).

⟨pos⟩ and ⟨decor⟩ are components of the special 'graphic language' which XY-pictures are specified in. We explain the language components in general terms in this § and in more depth in the following §§.

2.1 Positions

All *positions* may be written <X,Y> where X is the TeX dimension distance *right* and Y the distance *up* from the *zero position* 0 of the XY-picture (0 has coordinates <0mm,0mm>, of course). The zero position of the XY-picture determines the box produced by the \xy. . . \endxy command together with the four parameters $X_{\min}$, $X_{\max}$, $Y_{\min}$, and $Y_{\max}$ set such that all the objects in the picture are 'contained' in the following rectangle:

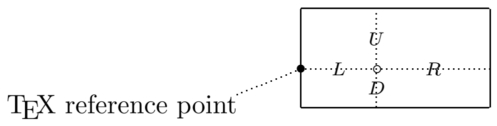


where the distances follow the "up and right > 0" principle, *e.g.*, the indicated TeX reference point has coordinates <$X_{\min}$,0pt> within the XY-picture. The zero position does not have to be contained in the picture, but $X_{\min} \le X_{\max} \wedge Y_{\min} \le Y_{\max}$ always holds. The possible positions are described in detail in §3.

When an XY-picture is entered in *math mode* then the reference point becomes the "vcenter" instead, *i.e.*, we use the point <$X_{\min}$,-\the\fontdimen22> as reference point.

2.2 Objects

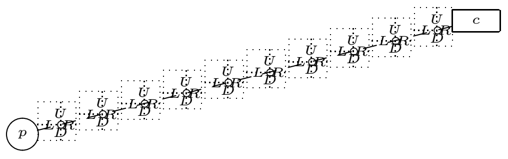
The simplest form of putting things into the picture is to 'drop' an *object* at a position. An object is like a TeX box except that it has a general *Edge* around its reference point—in particular this has the *extents* (*i.e.*, it is always contained within) the dimensions L, R, U, and D away from the reference point in each of the four directions left, right, up, and down. Objects are encoded in TeX boxes using the convention that the TeX reference point of an object is at its left edge, thus shifted <$-L$,0pt> from the center—so a TeX box may be said to be a rectangular object with $L =$ 0pt. Here is an example:



The object shown has a rectangle edge but others are available even though the kernel only supports rectangle and circle edges. It is also possible to use entire XY-pictures as objects with a rectangle edge, 0 as the reference point, $L = -X_{\min}$, $R = X_{\max}$, $D = -Y_{\min}$, and $U = Y_{\max}$. The commands for objects are described in §4.

2.3 Connections

Besides having the ability to be dropped at a position in a picture, all objects may be used to *connect* the two current objects of the state, *i.e.*, p and c. For most objects this is done by 'filling' the straight line between the centers with as many copies as will fit between the objects:



The ways the various objects connect are described along with the objects.

2.4 Decorations

When the \xy command reaches something that can not be interpreted as a continuation of the position being read, then it is expected to be a *decoration*, *i.e.*, in a restricted set of TeX commands which add to pictures. Most such commands are provided by the various *user options* (*cf.* §7)—only a few are provided within the kernel to facilitate programming of such options (and user macros) as described in §5.

2.5 The XY-pic state

Finally we summarise the user-accessible parts of the XY-picture state of two positions together with the last object associated with each: the *previous*, p, is the position <X_p, Y_p> with the object L_p, R_p, D_p, U_p, $Edge_p$, and the *current*, c, is the position <X_c, Y_c> with the object L_c, R_c, D_c, U_c, $Edge_c$.

Furthermore, XY-pic has a configurable *cartesian coordinate system* described by an *origin* position <X_{origin},Y_{origin}> and two *base vectors*

<X_{xbase},Y_{xbase}> and <X_{ybase},Y_{ybase}> accessed by the usual notation using parentheses:

$$(x,y) = < X_{origin} + x \times X_{xbase} + y \times X_{ybase}\ ,\quad Y_{origin} + x \times Y_{xbase} + y \times Y_{ybase} >$$

This is explained in full when we show how to set the base in note 3d of §3.

Finally typesetting a connection will setup a "placement state" for referring to positions on the connection that is accessed through a special `?` position construction; this is also discussed in detail in §3.

The XY-pic *state* consists of all these parameters together. They are initialised to zero except for $X_{xbase} = Y_{ybase} =$ `1mm`.

The edges are not directly available but points on the edges may be found using the different ⟨corner⟩ forms described in §3.

It is possible to insert an 'initial' piece of ⟨pos⟩ ⟨decor⟩ at the start of every XY-picture with the declaration

`\everyxy={` ⟨text⟩ `}`

This will act as if the ⟨text⟩ was typed literally right after each `\xy` command, parsing the actual contents as if it follows this – thus it is recommended that ⟨text⟩ has the form ⟨pos⟩`,` such that users can continue with ⟨pos⟩ ⟨decor⟩.

3 Positions

A ⟨pos⟩ition is a way of specifying locations as well as dropping objects at them and decorating them—in fact any aspect of the XY-pic state can be changed by a ⟨pos⟩ but most will just change the coordinates and/or shape of *c*.

All possible positions are shown in figure 1 with explanatory notes below.

Exercise 1: Which of the positions `0`, `<0pt,0pt>`, `<0pt>`, `(0,0)`, and `/0pt/` is different from the others?

Notes

3a. When doing arithmetic with `+` and `-` then the resulting current object inherits the size of the ⟨coord⟩, *i.e.*, the right argument—this will be zero if the ⟨coord⟩ is a ⟨vector⟩.

Exercise 2: How do you set *c* to an object the same size as the saved object `"ob"` but moved <X,Y>?

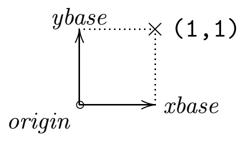
3b. *Skewing* using `!` just means that the reference point of *c* is moved with as little change to the shape of the object as possible, *i.e.*, the edge of *c* will remain in the same location except that it will grow larger to avoid moving the reference point outside *c*.

Exercise 3: What does the ⟨pos⟩ ... `!R-L` do?
Bug: The result of `!` is always a rectangle currently.

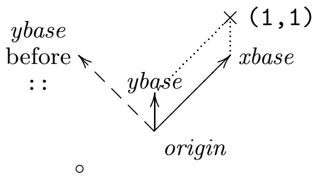
3c. A ⟨pos⟩ *covers* another if it is a rectangle with size sufficiently large that the other is "underneath". The `.` operation "extends" a ⟨pos⟩ to cover an additional one—the reference point of *c* is not moved but the shape is changed to a rectangle such that the entire *p* object is covered.

Bug: non-rectangular objects are first "translated" into a rectangle by using a diagonal through the object as the diagonal of the rectangle.

3d. The operations `:` and `::` set the *base* used for ⟨coord⟩inates having the form `(`x`,`y`)`. The `:` operation will set <X_{origin}, Y_{origin}> to *p*, <X_{xbase}, Y_{xbase}> to $c - origin$, and <X_{ybase}, Y_{ybase}> to <$-Y_{xbase}$, X_{xbase}> (this ensures that it is a usual square coordinate system). The `::` operation may then be used afterwards to make nonsqare bases by just setting *ybase* to $c - origin$. Here are two examples: firstly `0;<1cm,0cm>:` sets the coordinate system



while `<1cm,.5cm>;<2cm,1.5cm>:<1cm,1cm>::` defines



where in each case the ∘ is at `0`, the base vectors have been drawn and the × is at `(1,1)`.

When working with cartesian coordinates these three special ⟨factor⟩s are particularly useful:

`\halfroottwo`	$0.70710678 \approx \frac{1}{2}\sqrt{2}$
`\partroottwo`	$0.29289322 \approx 1 - \frac{1}{2}\sqrt{2}$
`\halfrootthree`	$0.86602540 \approx \frac{1}{2}\sqrt{3}$

More can be defined using `\def` (or `\newcommand` in LaTeX).

Syntax			Action
⟨pos⟩	$\longrightarrow$	⟨coord⟩	$c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `+` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ $+$ ⟨coord⟩$^{\text{3a}}$
	|	⟨pos⟩ `-` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ $-$ ⟨coord⟩$^{\text{3a}}$
	|	⟨pos⟩ `!` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ then skew$^{\text{3b}}$ c by ⟨coord⟩
	|	⟨pos⟩ `.` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ but also covering$^{\text{3c}}$ ⟨coord⟩
	|	⟨pos⟩ `,` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ then $c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `;` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩, swap p and c, $c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `:` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩, set base$^{\text{3d}}$, $c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `::` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩, $\mathit{ybase} \leftarrow c - \mathit{origin}$, $c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `*` ⟨object⟩	$c \leftarrow$ ⟨pos⟩, drop$^{\text{3f}}$ ⟨object⟩
	|	⟨pos⟩ `**` ⟨object⟩	$c \leftarrow$ ⟨pos⟩, connect$^{\text{3g}}$ using ⟨object⟩
	|	⟨pos⟩ `?` ⟨place⟩	$c \leftarrow$ ⟨pos⟩, $c \leftarrow$ ⟨place⟩$^{\text{3h}}$
	|	⟨pos⟩ `@` ⟨stacking⟩	$c \leftarrow$ ⟨pos⟩, do ⟨stacking⟩$^{\text{3o}}$
	|	⟨pos⟩ `=` ⟨saving⟩	$c \leftarrow$ ⟨pos⟩, do ⟨saving⟩$^{\text{3p}}$
⟨coord⟩	$\longrightarrow$	⟨vector⟩	⟨pos⟩ is ⟨vector⟩ with zero size
	|	⟨empty⟩ | `c`	reuse last c (do nothing)
	|	`p`	p
	|	`x` | `y`	axis intersection$^{\text{3k}}$ with $\overline{pc}$
	|	`s`⟨digit⟩ | `s{`⟨number⟩`}`	stack$^{\text{3o}}$ position ⟨digit⟩ or ⟨number⟩ below the top
	|	`"`⟨id⟩`"`	restore what was saved$^{\text{3p}}$ as ⟨id⟩ earlier
	|	`{` ⟨pos⟩ ⟨decor⟩ `}`	the c resulting from interpreting the group$^{\text{3l}}$
⟨vector⟩	$\longrightarrow$	`0`	zero
	|	`<` ⟨dimen⟩ `,` ⟨dimen⟩ `>`	absolute
	|	`<` ⟨dimen⟩ `>`	absolute with equal dimensions
	|	`(` ⟨factor⟩ `,` ⟨factor⟩ `)`	in current base$^{\text{3d}}$
	|	`a` `(` ⟨number⟩ `)`	angle in current base$^{\text{3e}}$
	|	⟨corner⟩	from reference point to ⟨corner⟩ of c
	|	⟨corner⟩ `(` ⟨factor⟩ `)`	The ⟨corner⟩ multiplied with ⟨factor⟩
	|	`/` ⟨direction⟩ ⟨dimen⟩ `/`	vector ⟨dimen⟩ in ⟨direction⟩$^{\text{3m}}$
⟨corner⟩	$\longrightarrow$	`L` | `R` | `D` | `U`	offset$^{\text{3n}}$ to left, right, down, up side
	|	`CL` | `CR` | `CD` | `CU` | `C`	offset$^{\text{3n}}$ to center of side, true center
	|	`LD` | `RD` | `LU` | `RU`	offset$^{\text{3n}}$ to actual left/down, … corner
	|	`E` | `P`	offset$^{\text{3n}}$ to nearest/proportional edge point to p
⟨place⟩	$\longrightarrow$	`<` ⟨place⟩ | `>` ⟨place⟩	shave$^{\text{3h}}$ `(0)/(1)` to edge of p/c, $f \leftarrow$ `0/1`
	|	`(` ⟨factor⟩ `)` ⟨place⟩	$f \leftarrow$ ⟨factor⟩
	|	⟨slide⟩	pick place$^{\text{3h}}$ and apply ⟨slide⟩
	|	`!` `{`⟨pos⟩`}` ⟨slide⟩	intercept$^{\text{3j}}$ with line setup by ⟨pos⟩ and apply ⟨slide⟩
⟨slide⟩	$\longrightarrow$	`/` ⟨dimen⟩ `/`	slide$^{\text{3i}}$ ⟨dimen⟩ further along connection
	|	⟨empty⟩	no slide

Figure 1: ⟨pos⟩itions.

3e. An *angle* α in XY-pic is the same as the coordinate pair `(`$\cos\alpha$`,`$\sin\alpha$`)` where α must be an integer interpreted as a number of degrees. Thus the ⟨vector⟩ `a(0)` is the same as `(1,0)` and `a(90)` as `(0,1)`, etc.

3f. To *drop* an ⟨object⟩ at c with `*` means to actually physically typeset it in the picture with reference position at c—how this is done depends on the ⟨object⟩ in question and is described in detail in §4. The intuition with a drop is that it typesets something at `<`X_c`,`Y_c`>` and sets the edge of c accordingly.

3g. The *connect* operation `**` will first compute a number of internal parameters describing the direction from p to c and then typesets a connection filled with copies of the ⟨object⟩ as illustrated in §2.3. The exact details of the connection depend on the actual ⟨object⟩ and are described in general in §4. The intuition with a connection is that it typesets something connecting p and c and sets the `?` ⟨pos⟩ operator up accordingly.

3h. Using `?` will "pick a place" along the most recent connection typeset with `**`. What exactly this means is determined by the object that was used for the connection and by the modifiers described in general terms here.

The "shave" modifiers in a ⟨place⟩, `<` and `>`, change the default ⟨factor⟩, f, and how it is used, by 'moving' the positions that correspond to `(0)` and `(1)` (respectively): These are initially set equal to p and c, but shaving will move them to the point on the edge of p and c where the connection "leaves/enters" them, and change the default f as indicated. When one end has already been shaved thus then subsequent shaves will correspond to sliding the appropriate position(s) a TEX `\jot` (usually equal to `3pt`) further towards the other end of the connection (and past it). Finally the *pick* action will pick the position located the fraction f of the way from `(0)` to `(1)` where f = `0.5` if it was not set (by `<`, `>`, or explicitly).

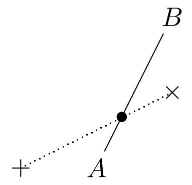
All this is probably best illustrated with some examples: each $\otimes$ in figure 2 is typeset by a sequence of the form p`;` c `**@{.} ?`⟨place⟩ `*{\oplus}` where we indicate the `?`⟨place⟩ in each case. (We also give examples of ⟨slide⟩s.)

3i. A ⟨slide⟩ will move the position a dimension further along the connection at the picked position. For straight connections (the only ones kernel XY-pic provides) this is the same as adding a vector in the tangent direction, *i.e.*, `?`...`/`A`/` is the same as `?`...`+/`A`/`.

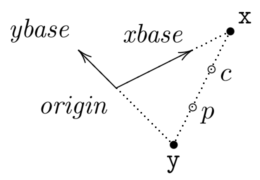
3j. This special ⟨place⟩ finds the point where the last connection intercepts with the line from p to c as setup by the ⟨pos⟩, thus usually this will have the form `!{`⟨coord⟩`;`⟨coord⟩`}`[5], for example, **Bug:** Only works for straight arrows at present.

```
\xy <1cm,0cm>:
 (0,0)*=0{+}="+" ;
 (2,1)*=0{\times}="*" **@{.} ,
 (1,0)*+{A} ; (2,2)*+{B} **@{-}
 ?!{"+";"*"} *{\bullet}
\endxy
```

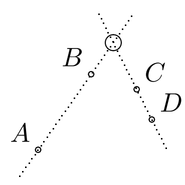
will typeset



3k. The positions denoted by the *axis intersection* ⟨coord⟩inates `x` and `y` are the points where the line through p and c intersects with each axis. The following figure illustrates this:



Exercise 4: Given predefined points A, B, C, and D (stored as objects `"A"`, `"B"`, `"C"`, and `"D"`), write a ⟨coord⟩ specification that will return the point where the lines $\overline{AB}$ and $\overline{CD}$ cross (the point marked with a large circle here):



3l. A ⟨pos⟩ ⟨decor⟩ *grouped* in `{}`-braces[6] is interpreted in a local scope in the sense that any p and *base* built within it are forgotten afterwards, leaving only the c as the result of the ⟨coord⟩. **Note:** Only p and *base* are restored – it is not a TEX group.

[5]The braces can be replaced by `(*`...`*)` *once*, *i.e.*, there can be no other braces nested inside it.
[6]One can use `(*`...`*)` instead also here.

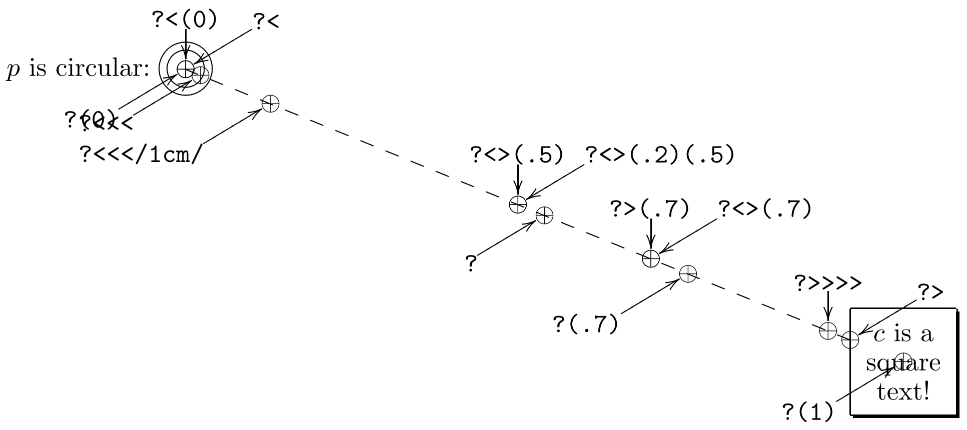


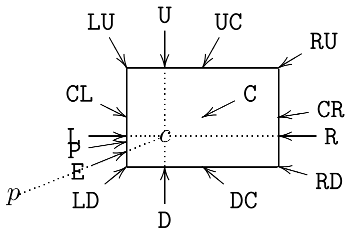
Figure 2: Example ⟨place⟩s

Exercise 5: What effect is achieved by using the ⟨coord⟩inate "`{;}`"?

3m. The vector `/`Z`/`, where Z is a ⟨dimen⟩sion, is the same as the vector `<`$Z\cos\alpha$`,`$Z\sin\alpha$`>` where α is the angle of the last direction set by a connection (*i.e.*, with `**`) or subsequent placement (`?`) position.

It is possible to give a ⟨direction⟩ as described in the next section (figure 3, note 4l in particular) that will then be used to set the value of α. It is also possible to omit the ⟨dimen⟩ in which case it is set to a default value of `.5pc`.

3n. A ⟨corner⟩ is an offset from the current `<`X_c`,`Y_c`>` position to a specific position on the edge of the c object (the two-letter ones may be given in any combination):



The 'edge point' `E` lies on the edge along the line from p to the centre of the object, in contrast to the 'proportional' point `P` which is also a point on the edge but computed in such a way that the object looks as much 'away from p' as possible.

Finally, a following `(`f`)` suffix will multiply the offset vector by the ⟨factor⟩ f.

Exercise 6: What is the difference between the ⟨pos⟩itions `c?<` and `c+E`?

Exercise 7: What does this typeset?

```
\xy *=<3cm,1cm>\txt{Box}*\frm{-}
 !U!R(.5) *\frm{..}*{\bullet} \endxy
```

Hint: `\frm` is defined by the frame extension and just typesets a frame of the kind indicated by the argument.

Bug: Currently only the single-letter corners (`L`, `R`, `D`, `U`, `C`, `E`, and `P`) will work for any shape—the others silently assume that the shape is rectangular.

3o. The *stack* is a special construction useful for storing a sequence of ⟨pos⟩itions that are accessible using the special ⟨coord⟩inates `s`n, where n is either a single digit or a positive integer in `{}`s: `s0` is always the 'top' element of the stack and if the stack has depth d then the 'bottom' element of the stack has number `s{`$d-1$`}`. The stack is said to be 'empty' when the depth is 0 and then it is an error to access any of the `s`n or 'pop' which means remove the top element, shifting what is in `s1` to `s0`, `s2` to `s1`, etc. Similarly, 'push c' means to shift `s0` to `s1`, etc., and then insert the c as the new `s0`.

The stack is manipulated as follows:

`@`⟨stacking⟩	Action
`@+`⟨coord⟩	push ⟨coord⟩
`@-`⟨coord⟩	$c \leftarrow$ ⟨coord⟩ then pop
`@=`⟨coord⟩	load stack with ⟨coord⟩
`@@`⟨coord⟩	do ⟨coord⟩ for $c \leftarrow$ stack
`@i`	initialise
`@(`	enter new frame
`@)`	leave current frame

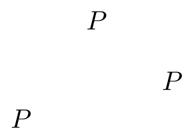
To 'load stack', means to load the entire stack with the positions set by ⟨coord⟩ within which `,` means 'push c'.

To 'do ⟨coord⟩ for all stack elements' means to set c to each element of the stack in turn, from the bottom and up, and for each interpret the ⟨coord⟩. Thus the first interpretation has c set to the bottom element of the stack and the last has c set to `s0`. If the stack is empty, the ⟨coord⟩ is not interpreted at all.

These two operations can be combined to repeat a particular ⟨coord⟩ for several points, like this:

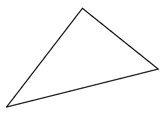
```
\xy
 @={(0,-10),(10,3),(20,-5)} @@{*{P}}
\endxy
```

will typeset



Finally, the stack can be forcibly cleared using `@i`, however, this is rarely needed because of `@(`, which saves the stack as it is, and then clears it, such when it has been used (and is empty), and `@)` is issued, then it is restored as it was at the time of the `@(`.

Exercise 8: How would you change the example above to connect the points as shown below?



3p. It is possible to define new ⟨coord⟩inates on the form `"`⟨id⟩`"` by *saving* the current c using the ...`="`⟨id⟩`"` ⟨pos⟩ition form. Subsequent uses of `"`⟨id⟩`"` will then reestablish the c at the time of the saving.

Using a `"`⟨id⟩`"` that was never defined is an error, however, saving into a name that was previously defined just replaces the definition without warning, *i.e.*, `"`⟨id⟩`"` always refers to the last thing saved with that ⟨id⟩.

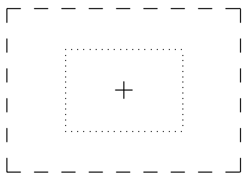
However, many other things can be 'saved': in general `@`⟨saving⟩ has either of the forms

`@:"`⟨id⟩`"`	`"`⟨id⟩`"` restores current *base*
`@`⟨coord⟩`"`⟨id⟩`"`	`"`⟨id⟩`"` reinterprets ⟨coord⟩
`@@"`⟨id⟩`"`	`@="`⟨id⟩`"` reloads this stack

The first form defines `"`⟨id⟩`"` to be a macro that restores the current *base*.

The second does not depend on the state at the time of definition at all; it is a macro definition. You can pass parameters to such a macro by letting it use coordinates named `"1"`, `"2"`, etc., and then use `="1"`, `="2"`, etc., just before every use of it to set the actual values of these. **Note:** it is not possible to use a ⟨coord⟩ of the form `"`⟨id⟩`"` directly: write it as `{"`⟨id⟩`"}`.

Exercise 9: Write a macro `"dbl"` to double the size of the current c object, *e.g.*, changing it from the dotted to the dashed outline in this figure:



The final form defines a special kind of macro that should only be used after the `@=` stack operation: the entire current stack is saved such that the stack operation `@="`⟨id⟩`"` will reload it.

Note: There is no distinction between the 'name spaces' of ⟨id⟩s used for saved coordinates and other things.

4 Objects

Objects are the entities that are manipulated with the `*` and `**` ⟨pos⟩ operations above to actually get some output in XY-pictures. As for ⟨pos⟩itions the operations are interpreted strictly from left to right, however, the actual object is built *before* all the ⟨modifier⟩s take effect. The syntax of objects is given in figure 3 with references to the notes below. **Remark:** It is *never* allowed to include braces `{}` inside ⟨modifier⟩s! In case you wish to do something that requires `{`...`}` then check in this manual whether you can use `(*`...`*)` instead. If not then you will have to use a different construction!

Notes

4a. An ⟨object⟩ is built using `\objectbox {`⟨text⟩`}`. `\objectbox` is initially defined as

```
\def\objectbox#1{%
 \hbox{$\objectstyle{#1}$}}
\let\objectstyle=\displaystyle
```

but may be redefined by options or the user. The ⟨text⟩ should thus be in the mode required by the `\objectbox` command—with the default `\objectbox` shown above it should be in math mode.

Syntax			Action
⟨object⟩	⟶	⟨modifier⟩ ⟨object⟩	apply ⟨modifier⟩ to ⟨object⟩
	|	⟨objectbox⟩	build ⟨objectbox⟩ then apply its ⟨modifier⟩s
⟨objectbox⟩	⟶	`{` ⟨text⟩ `}`	build default[4a] object
	|	⟨library object⟩ | `@`⟨dir⟩	use ⟨library object⟩ or ⟨dir⟩ectional (see §6)
	|	⟨TeX box⟩ `{` ⟨text⟩ `}`	build box[4b] object with ⟨text⟩ using the given ⟨TeX box⟩ command, *e.g.*, `\hbox`
	|	`\object` ⟨object⟩	wrap up the ⟨object⟩ as a finished object box[4c]
	|	`\composite {` ⟨composite⟩ `}`	build composite object box[4d]
	|	`\xybox {` ⟨pos⟩ ⟨decor⟩ `}`	package entire XY-picture as object[4e]
⟨modifier⟩	⟶	`!` ⟨vector⟩	⟨object⟩ has its reference point shifted[4f] by ⟨vector⟩
	|	`!`	⟨object⟩ has the original reference point reinstated
	|	⟨add op⟩ ⟨size⟩	change ⟨object⟩ size[4g]
	|	`h` | `i`	⟨object⟩ is hidden[4h], invisible[4i]
	|	`[` ⟨shape⟩ `]`	⟨object⟩ is given the specified ⟨shape⟩[4j]
	|	`[=` ⟨shape⟩ `]`	define ⟨shape⟩[4k] to reestablish current object style
	|	⟨direction⟩	set current direction for this ⟨object⟩
⟨add op⟩	⟶	`+` | `-` | `=` | `+=` | `-=`	grow, shrink, set, grow to, shrink to
⟨size⟩	⟶	⟨empty⟩	default size[4g]
	|	⟨vector⟩	size as sides of rectangle covering the ⟨vector⟩
⟨direction⟩	⟶	⟨diag⟩	⟨diag⟩onal direction[4l]
	|	`v` ⟨vector⟩	direction[4l] of ⟨vector⟩
	|	`q{` ⟨pos⟩ ⟨decor⟩ `}`	direction[4l] from p to c after ⟨pos⟩ ⟨decor⟩
	|	⟨direction⟩ `:` ⟨vector⟩	vector relative to ⟨direction⟩[4m]
	|	⟨direction⟩ `_` | ⟨direction⟩ `^`	90° clockwise/anticlockwise to ⟨direction⟩[4m]
⟨diag⟩	⟶	⟨empty⟩	last used direction (not necessarily diagonal[4l])
	|	`l` | `r` | `d` | `u`	left, right, down, up diagonal[4l]
	|	`ld` | `rd` | `lu` | `ru`	left/down, . . . diagonal[4l]
⟨composite⟩	⟶	⟨object⟩	first object is required
	|	⟨composite⟩ `*` ⟨object⟩	add ⟨object⟩ to composite object box[4d]

Figure 3: ⟨object⟩s.

4b. An ⟨object⟩ built from a TEX box with dimensions $w \times (h+d)$ will have $L_c = R_c = w/2$, $U_c = D_c = (h+d)/2$, thus initially be equipped with the adjustment `!C` (see note 4f). In particular: in order to get the reference point on the (center of) the base line of the original ⟨TEX box⟩ then you should use the ⟨modifier⟩ `!`; to get the reference point identical to the TEX reference point use the modifier `!!L`.

TEXnical remark: Any macro that expands to something that starts with a ⟨box⟩ may be used as a ⟨TEX box⟩ here.

4c. Takes an object and constructs it, building a box; it is then processed according to the preceeding modifiers. This form makes it possible to use any ⟨object⟩ as a TEX box (even outside of XY-pictures) because a finished object is always also a box.

4d. Several ⟨object⟩s can be combined into a single object using the special command `\composite` with a list of the desired objects separated with `*`s as the argument. The resulting box (and object) is the least rectangle enclosing all the included objects.

4e. Take an entire XY-picture and wrap it up as a box as described in §2.1. Makes nesting of XY-pictures possible: the inner picture will have its own zero point which will be its reference point *in* the outer picture when it is placed there.

4f. An object is *shifted* a ⟨vector⟩ by moving the point inside it which will be used as the reference point. This effectively pushes the object the same amount in the opposite direction.

Exercise 10: What is the difference between the ⟨pos⟩itions `0*{a}!DR` and `0*!DR{a}`?

4g. A ⟨size⟩ is a pair `<`W`,`H`>` of the width and height of a rectangle. When given as a ⟨vector⟩ these are just the vector coordinates, *i.e.*, the ⟨vector⟩ starts in the lower left corner and ends in the upper right corner. The possible ⟨add op⟩erations that can be performed are described in the following table.

⟨add op⟩	description
`+`	grow
`-`	shrink
`=`	set to
`+=`	grow to at least
`-=`	shrink to at most

In each case the ⟨vector⟩ may be omitted which invokes the "default size" for the particular ⟨add op⟩:

⟨add op⟩	default
`+`	`+<`$2 \times$ *objectmargin*`>`
`-`	`-<`$2 \times$ *objectmargin*`>`
`=`	`=<`*objectwidth*`,`*objectheight*`>`
`+=`	`+=<` $\max(L_c + R_c, D_c + U_c)$`>`
`-=`	`-=<` $\min(L_c + R_c, D_c + U_c)$`>`

The defaults for the first three are set with the commands

`\objectmargin` ⟨add op⟩ `{`⟨dimen⟩`}`
`\objectwidth` ⟨add op⟩ `{`⟨dimen⟩`}`
`\objectheight` ⟨add op⟩ `{`⟨dimen⟩`}`

where ⟨add op⟩ is interpreted in the same way as above.

The defaults for `+=`/`-=` are such that the resulting object will be the smallest containing/largest contained square.

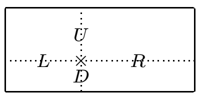
Exercise 11: How are the objects typeset by the ⟨pos⟩itions "`*+UR{\sum}`" and "`*+DL{\sum}`" enlarged?

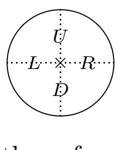
Bug: Currently changing the size of a circular object is buggy—it is changed as if it is a rectangle and then the change to the R parameter affects the circle. This should be fixed probably by a generalisation of the `o` shape to be ovals or ellipses with horizontal/vertical axes.

4h. A *hidden* object will be typeset but hidden from XY-pic in that it won't affect the size of the entire picture as discussed in §2.1.

4i. An *invisible* object will be treated completely normal except that it won't be typeset, *i.e.*, XY-pic will behave as if it was.

4j. Setting the *shape* of an object forces the shape of its edge to be as indicated. The kernel provides three shapes that change the edge, namely `[.]`, `[]`, and `[o]`, corresponding to the outlines

$\times$,


, and


where the $\times$ denotes the point of the reference position in the object (the first is a point). Extensions can provide more shapes, however, all shapes set the extent dimensions L, R, D, and U.

The default shape for objects is `[]` and for plain coordinates it is `[.]`.

Furthermore the ⟨shape⟩s `[r]`, `[l]`, `[u]`, and `[d]`, are defined for convenience to adjust the object to the indicated side by setting the reference point such that the reference point is the same distance from the opposite of the indicated edge and the two neighbour edges but never closer to the indicated side than the opposite edge, *e.g.*, the object `[r]\hbox{Wide text}` has reference point at the × in $\boxed{\times\text{Wide text}}$ but the object `[d]\hbox{Wide text}` has reference point at the × in $\boxed{\text{Wide}\times\text{text}}$. Finally, `[c]` puts the reference point at the center.

Note: Extensions can add new ⟨shape⟩ object ⟨modifier⟩s which are then called ⟨style⟩s. These will always be either of the form `[`⟨keyword⟩`]` or `[`⟨character⟩ ⟨argument⟩`]`. Some of these ⟨style⟩s do other things than set the edge of the object.

4k. While typesetting an object, some of the properties are considered part of the 'current object style'. Initially this means nothing but some of the ⟨style⟩s defined by extensions have this status, *e.g.*, colours `[red]`, `[blue]` say, using the `xycolor` extension, or varying the width of lines using `xyline`. Such styles are processed *left-to-right*; for example,

```
*[red][green][=NEW][blue]{A}
```

will typeset a blue `A` and define `[NEW]` to set the colour to green (all provided that `xycolor` has been loaded, of course).

Saving styles: Once specified for an ⟨object⟩, the collection of ⟨style⟩s can be assigned a name, using `[=`⟨word⟩`]`. Then `[`⟨word⟩`]` becomes a new ⟨style⟩, suitable for use with the same or other ⟨objects⟩s. Use a single ⟨word⟩ built from ordinary letters. If `[`⟨word⟩`]` already had meaning the new definition will still be imposed, but the following type of warning will be issued:

```
Xy-pic Warning: Redefining style [⟨word⟩]
```

The latter warning will appear if the definition occurs within an `\xymatrix`. This is perfectly normal, being a consequence of the way that the matrix code is handled. Similarly the message may appear several times if the style definition is made within an `\ar`.

The following illustrates how to avoid these messages by defining the style without typesetting anything.

```
\setbox0=\hbox{%
 \xy\drop[OrangeRed][=A]{}\endxy}
```

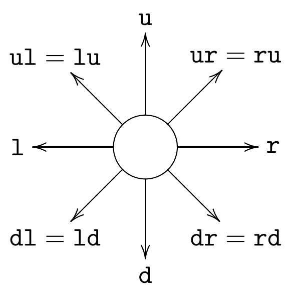
Note 1: The current colour is regarded as part of the style for this purpose.

Note 2: Such namings are global in scope. They are intended to allow a consistent style to be easily maintained between various pictures and diagrams within the same document.

If the same ⟨style⟩ is intended for several ⟨object⟩s occurring in succession, the `[|*]` ⟨modifier⟩ can be used on the later ⟨object⟩s. This only works when `[|*]` precedes any other ⟨style⟩ modifiers; it is local in scope, recovering the last ⟨style⟩s used at the same level of TeX grouping.

4l. Setting the current direction is simply pretending for the typesetting of the object (and the following ⟨modifier⟩s) that some connection set it – the ⟨empty⟩ case just inherits the previous direction.

It is particularly easy to set ⟨diag⟩onal directions:



Alternatively `v`⟨vector⟩ sets the direction as if the connection from `0` to the ⟨vector⟩ had been typeset except that the *origin* is assumed zero such that directions `v(`x`,`y`)` mean the natural thing, *i.e.*, is the direction of the connection from `(0,0)` to `(`x`,`y`)`.

In case the direction is not as simple, you can construct `{` ⟨pos⟩ ⟨decor⟩ `}` that sets up p and c such that $\overline{pc}$ has the desired direction. **Note:** that you must use the `(*...*)` form if this is to appear in an object ⟨modifier⟩!

Exercise 12: What effect is achieved by using ⟨modifier⟩s `v/1pc/` and `v/-1pc/`?

4m. Once the initial direction is established as either the last one or an absolute one then the remainder of the ⟨direction⟩ is interpreted.

Adding a single `^` or `_` denotes the result of rotating the default direction a right angle in the positive and negative direction, *i.e.*, anti-/clockwise, respectively. **Note:** Do *not* use `^^` but only `__` to reverse the direction!

A trailing `:`⟨vector⟩ is like `v`⟨vector⟩ but uses the ⟨direction⟩ to set up a standard square base such that `:(0,1)` and `:(0,-1)` mean the same as `:a(90)` and `:a(-90)` and as `^` and `_`, respectively.

Exercise 13: What effect is achieved by using ⟨modifier⟩s `v/1pc/:(1,0)` and `v/-1pc/__`?

5 Decorations

⟨Decor⟩ations are actual TeX macros that decorate the current picture in manners that depend on the state. They are allowed *after* the ⟨pos⟩ition either of the outer `\xy`...`\endxy` or inside `{`...`}`. The possibilities are given in figure 4 with notes below.

Most options add to the available ⟨decor⟩, in particular the `v2` option loads many more since Xy-pic versions prior to 2.7 provided most features as ⟨decor⟩.

Notes

5a. Saving and restoring allows 'excursions' where lots of things are added to the picture without affecting the resulting Xy-pic state, *i.e.*, *c*, *p*, and *base*, and without requiring matching `{}`s. The independence of `{}` is particularly useful in conjunction with the `\afterPOS` command, for example, the definition

```
\def\ToPOS{\save\afterPOS{%
  \POS**{}?>*@2{>}**@{-}\restore};p,}
```

will cause the code `\ToPOS`⟨pos⟩ to construct a double-shafted arrow from the current object to the ⟨pos⟩ (computed relative to it) such that `\xy *{A} \ToPOS +<10mm,2mm>\endxy` will typeset the picture .

Note: Saving this way in fact uses the same state as the `{}` 'grouping', so the code p_1`,` `{`p_2`\save}`, ... `{\restore}` will have $c = p_1$ both at the ... and at the end!

5b. One very tempting kind of TeX commands to perform as ⟨decor⟩ is arithmetic operations on the Xy-pic state. This will work in simple Xy-pictures as described here but be warned: *it is not portable* because all Xy-pic execution is indirect, and this is used by several options in non-trivial ways. Check the TeX-nical documentation [15] for details about this!

Macros that expand to ⟨decor⟩ will always do the same, though.

5c. `\xyecho` will turn on echoing of all interpreted Xy-pic ⟨pos⟩ characters. **Bug:** Not completely implemented yet. `\xyverbose` will switch on a tracing of all Xy-pic commands executed, with line numbers. `\xytracing` traces even more: the entire Xy-pic state is printed after each modification. `\xyquiet` restores default quiet operation.

5d. Ignoring means that the ⟨pos⟩ ⟨decor⟩ is still parsed the usual way but nothing is typeset and the Xy-pic state is not changed.

5e. It is possible to save an intermediate form of commands that generate parts of an Xy-picture to a file such that subsequent typesetting of those parts is significantly faster: this is called *compiling*. The produced file contains code to check that the compiled code still corresponds to the same ⟨pos⟩⟨decor⟩ as well as efficient Xy-code to redo it; if the ⟨pos⟩⟨decor⟩ has changed then the compilation is redone.

There are two ways to use this. The direct is to invent a ⟨name⟩ for each diagram and then embrace it in `\xycompileto{`⟨name⟩`}|{`...`}` – this dumps the compiled code into the file ⟨name⟩`.xyc`.

When many diagrams are compiled then it is easier to add `\xycompile{`...`}` around the ⟨pos⟩⟨decor⟩ to be compiled. This will assign file names numbered consecutively with a ⟨prefix⟩ which is initially the expansion of `\jobname-` but may be set with

```
\CompilePrefix{⟨prefix⟩}
```

This has the disadvantage, however, that if additional compiled Xy-pictures are inserted then all subsequent pictures will have to be recompiled. One particular situation is provided, though: when used within constructions that typeset their contents more than once (such as most AMS-LaTeX equation constructs) then the declaration

```
\CompileFixPoint{⟨id⟩}
```

can be used inside the environment to fix the counter to have the same value at every passage.

Finally, when many 'administrative typesetting runs' are needed, *e.g.*, readjusting LaTeX cross references and such, then it may be an advantage to not typeset any Xy-pictures at all during the intermediate runs. This is supported by the following declarations which for each compilation creates a special file with the extension `.xyd` containing just the size of the picture:

```
\MakeOutlines
\OnlyOutlines
\ShowOutlines
```

Syntax			Action
⟨decor⟩	⟶	⟨command⟩ ⟨decor⟩	either there is a command...
	|	⟨empty⟩	...or there isn't.
⟨command⟩	⟶	`\save` ⟨pos⟩	save state[5a], then do ⟨pos⟩
	|	`\restore`	restore state[5a] saved by matcing `\save`
	|	`\POS` ⟨pos⟩	interpret ⟨pos⟩
	|	`\afterPOS {` ⟨decor⟩ `}` ⟨pos⟩	interpret ⟨pos⟩ and then perform ⟨decor⟩
	|	`\drop` ⟨object⟩	drop ⟨object⟩ as the ⟨pos⟩ `*` operation
	|	`\connect` ⟨object⟩	connect with ⟨object⟩ as the ⟨pos⟩ `**` operation
	|	`\relax`	do nothing
	|	⟨TeX commands⟩	any TeX commands[5b] and user-defined macros that neither generates output (watch out for stray spaces!), nor changes the grouping, may be used
	|	`\xyverbose` | `\xytracing` | `\xyquiet`	tracing[5c] commands
	|	`\xyignore {`⟨pos⟩ ⟨decor⟩`}`	ignore[5d] Xy-code
	|	`\xycompile {`⟨pos⟩ ⟨decor⟩`}`	compile[5e] to file ⟨prefix⟩⟨no⟩`.xyc`
	|	`\xycompileto{`⟨name⟩`}{`⟨pos⟩⟨decor⟩`}`	compile[5e] to file ⟨name⟩`.xyc`

Figure 4: ⟨decor⟩ations.

```
\NoOutlines
```

The first does no more. The second uses the file to typesets a dotted frame of the appropriate size instead of the picture (unless the picture has changed and is recompiled, then it is typeset as always and the `.xyd` file is recreated for subsequent runs). The third shows the outlines as dotted rectangles. The last switches outline processing completely off.

6 Kernel object library

In this section we present the *library objects* provided with the kernel language—several options add more library objects. They fall into three types: Most of the kernel objects (including all those usually used with `**` to build connections) are *directionals*, described in §6.1. The remaining kernel library objects are *circles* of §6.2 and *text* of §6.3.

6.1 Directionals

The kernel provides a selection of *directionals*: objects that depend on the current direction. They all take the form

`\dir`⟨dir⟩

to typeset a particular ⟨dir⟩ectional object. All have the structure

⟨dir⟩ ⟶ ⟨variant⟩`{`⟨main⟩`}`

with ⟨variant⟩ being ⟨empty⟩ or one of the characters `^_23` and ⟨main⟩ some mnemonic code.

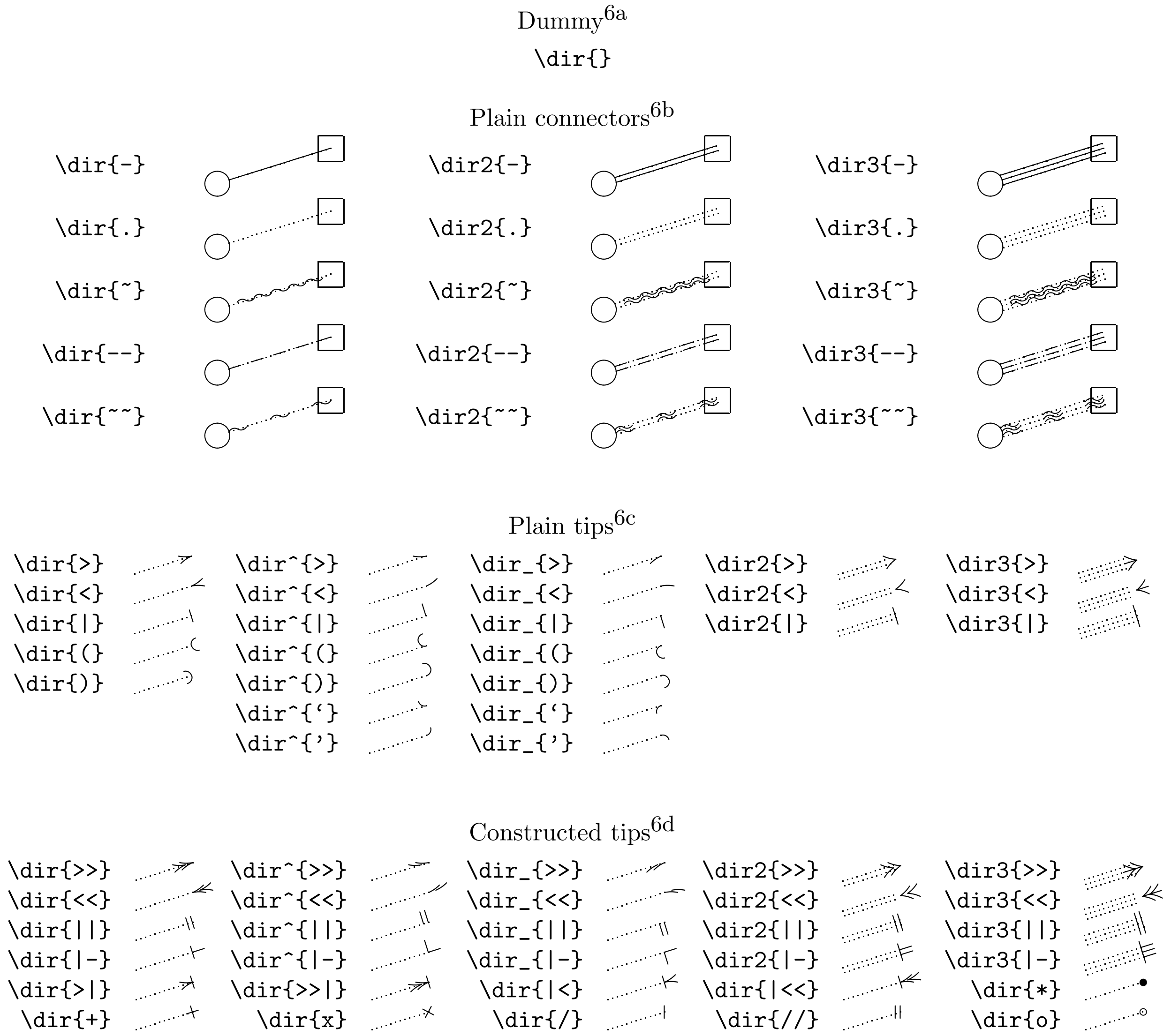
We will classify the directionals primarily intended for building connections as *connectors* and those primarily intended for placement at connection ends or as markers as *tips*.

Figure 5 shows all the ⟨dir⟩ectionals defined by the kernel with notes below; each ⟨main⟩ type has a line showing the available ⟨variant⟩s. Notice that only some variants exist for each ⟨dir⟩—when a nonexisting variant of a ⟨dir⟩ is requested then the ⟨empty⟩ variant is used silently. Each is shown in either of the two forms available in each direction as applicable: connecting a ◯ to a □ (typeset by `**\dir`⟨dir⟩) and as a tip at the end of a dotted connection of the same variant (*i.e.*, typeset by the ⟨pos⟩ `**\dir`⟨variant⟩`{.} ?> *\dir`⟨dir⟩).

As a special case an entire ⟨object⟩ is allowed as a ⟨dir⟩ by starting it with a `*`: `\dir*` is equivalent to `\object`.

Notes

6a. You may use `\dir{}` for a "dummy" directional object (in fact this is used automatically by `**{}`). This is useful for a uniform treatment of connections, *e.g.*, making the `?` ⟨pos⟩ able to find a point on the straight line from p to c without actually typesetting anything.



Dummy[6a]

```
\dir{}
```

Plain connectors[6b]

```
\dir{-}     \dir2{-}     \dir3{-}
\dir{.}     \dir2{.}     \dir3{.}
\dir{~}     \dir2{~}     \dir3{~}
\dir{--}    \dir2{--}    \dir3{--}
\dir{~~}    \dir2{~~}    \dir3{~~}
```

Plain tips[6c]

```
\dir{>}    \dir^{>}    \dir_{>}    \dir2{>}    \dir3{>}
\dir{<}    \dir^{<}    \dir_{<}    \dir2{<}    \dir3{<}
\dir{|}    \dir^{|}    \dir_{|}    \dir2{|}    \dir3{|}
\dir{(}    \dir^{(}    \dir_{(}
\dir{)}    \dir^{)}    \dir_{)}
           \dir^{`}    \dir_{`}
           \dir^{'}    \dir_{'}
```

Constructed tips[6d]

```
\dir{>>}    \dir^{>>}    \dir_{>>}    \dir2{>>}    \dir3{>>}
\dir{<<}    \dir^{<<}    \dir_{<<}    \dir2{<<}    \dir3{<<}
\dir{||}    \dir^{||}    \dir_{||}    \dir2{||}    \dir3{||}
\dir{|-}    \dir^{|-}    \dir_{|-}    \dir2{|-}    \dir3{|-}
\dir{>|}    \dir{>>|}    \dir{|<}     \dir{|<<}    \dir{*}
\dir{+}     \dir{x}      \dir{/}      \dir{//}     \dir{o}
```

Figure 5: Kernel library ⟨dir⟩ectionals

6b. The *plain connectors* group contains basic directionals that lend themself to simple connections.

By default XY-pic will typeset horizontal and vertical `\dir{-}` connections using TEX rules. Unfortunately rules is the feature of the DVI format most commonly handled wrong by DVI drivers. Therefore XY-pic provides the ⟨decor⟩ations

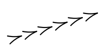
```
\NoRules
\UseRules
```

that will switch the use of such off and on.

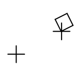
As can be seen by the last two columns, these (and most of the other connectors) also exist in double and triple versions with a `2` or a `3` prepended to the name. For convenience `\dir{=}` and `\dir{:}` are synonyms for `\dir2{-}` and `\dir2{.}`, respectively; similarly `\dir{==}` is a synonym for `\dir2{--}`.

6c. The group of *plain tips* contains basic objects that are useful as markers and arrowheads making connections, so each is shown at the end of a dotted connection of the appropriate kind.

They may also be used as connectors and will build dotted connections. *e.g.*, `**@{>}` typesets



Exercise 14: Typeset the following two +s and a tilted square:



Hint: the dash created by `\dir{-}` has the length `5pt` (here).

6d. These tips are combinations of the plain tips provided for convenience (and optimised for efficiency). New ones can be constructed using `\composite` and by declarations of the form

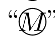
```
\newdir ⟨dir⟩ {⟨composite⟩}
```

which defines `\dir`⟨dir⟩ as the ⟨composite⟩ (see note 4d for the details).

6.2 Circle segments

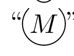
Circle ⟨object⟩s are round and typeset a segment of the circle centered at the reference point. The syntax of circles is described in figure 6 with explanations below.

The default is to generate a *full circle* with the specified radius, *e.g.*,

`\xy*\cir<4pt>{}\endxy` typesets " "
`\xy*{M}*\cir{}\endxy` — ""

All the other circle segments are subsets of this and have the shape that the full circle outlines.

Partial circle segments with ⟨orient⟩ation are the part of the full circle that starts with a tangent vector in the direction of the first ⟨diag⟩onal (see note 4l) and ends with a tangent vector in the direction of the other ⟨diag⟩onal after a clockwise (for `_`) or anticlockwise (for `^`) turn, *e.g.*,

`\xy*\cir<4pt>{l^r}\endxy` typesets " "
`\xy*\cir<4pt>{l_r}\endxy` — " "
`\xy*\cir<4pt>{dl^u}\endxy` — " "
`\xy*\cir<4pt>{dl_u}\endxy` — " "
`\xy*+{M}*\cir{dr_ur}\endxy` — ""

If the same ⟨diag⟩ is given twice then nothing is typeset, *e.g.*,

`\xy*\cir<4pt>{u^u}\endxy` typesets " "

Special care is taken to setup the ⟨diag⟩onal defaults:

- After `^` the default is the diagonal 90° anticlockwise from the one before the `^`.
- After `_` the default is the diagonal 90° clockwise from the one before the `_`.

The ⟨diag⟩ before `^` or `_` is required for `\cir` ⟨objects⟩.

Exercise 15: Typeset the following shaded circle with radius `5pt`:

6.3 Text

Text in pictures is supported through the ⟨object⟩ construction

```
\txt ⟨width⟩ ⟨style⟩ {⟨text⟩}
```

that builds an object containing ⟨text⟩ typeset to ⟨width⟩ using ⟨style⟩; in ⟨text⟩ `\\` can be used as an explicit line break; all lines will be centered. ⟨style⟩ should either be a font command or some other stuff to do for each line of the ⟨text⟩ and ⟨width⟩ should be either `<`⟨dimen⟩`>` or ⟨empty⟩.

7 XY-pic options

Note: LATEX 2$_\varepsilon$ users should also consult the paragraph on "xy.sty" in §1.1.

Syntax			Action
`\cir` ⟨radius⟩ `{` ⟨cir⟩ `}`			⟨cir⟩cle segment with ⟨radius⟩
⟨radius⟩	$\longrightarrow$	⟨empty⟩	use R_c as the radius
	|	⟨vector⟩	use X of the ⟨vector⟩ as radius
⟨cir⟩	$\longrightarrow$	⟨empty⟩	full circle of ⟨radius⟩
	|	⟨diag⟩ ⟨orient⟩ ⟨diag⟩	partial circle from first ⟨diag⟩onal through to the second ⟨diag⟩onal in the ⟨orient⟩ation
⟨orient⟩	$\longrightarrow$	`^`	anticlockwise
	|	`_`	clockwise

Figure 6: ⟨cir⟩cles.

7.1 Loading

XY-pic is provided with a growing number of options supporting specialised drawing tasks as well as exotic output devices with special graphic features. These should all be loaded using this uniform interface in order to ensure that the XY-pic environment is properly set up while reading the option.

```
\xyoption { ⟨option⟩ }
\xyrequire { ⟨option⟩ }
```

`\xyoption` will cause the loading of an XY-pic option file which can have one of several names. These are tried in sequence: `xy`⟨option⟩`.tex`, `xy`⟨option⟩`.doc`, `xy`⟨short⟩`.tex`, and `xy`⟨short⟩`.doc`, where ⟨short⟩ is ⟨option⟩ truncated to 6 (six) characters to conform with the TWG-TDS [17].

`\xyrequire` is the same except it is ignored if an option with the same name is already present (thus does not check the version etc.).

Sometimes some declarations of an option or header file or whatever only makes sense after some particular other option is loaded. In that case the code should be wrapped in the special command

```
\xywithoption { ⟨option⟩ } { ⟨code⟩ }
```

which indicates that if the ⟨option⟩ is already loaded then ⟨code⟩ should be executed now, otherwise it should be saved and if ⟨option⟩ ever gets loaded then ⟨code⟩ should be executed afterwards. **Note:** The ⟨code⟩ should allow more than one execution; it is saved with the catcodes at the time of the `\xywithoption` command.

Finally, it is possible to declare ⟨code⟩ as some commands to be executed before every actual execution of `\xywithoption{`⟨option⟩`}{`...`}`, and similarly ⟨code⟩ to be executed before every `\xyoption{`⟨option⟩`}` and `\xyrequire{`⟨option⟩`}` (collectively called 'requests')::

```
\xyeverywithoption { ⟨option⟩ } { ⟨code⟩ }
\xyeveryrequest { ⟨option⟩ } { ⟨code⟩ }
```

This is most often used by an option to activate some hook every time it is requested itself.

7.2 Option file format

Option files must have the following structure:

```
%% ⟨identification⟩
%% ⟨copyright, etc.⟩
\ifx\xyloaded\undefined \input xy \fi
\xyprovide{⟨option⟩}{⟨name⟩}{⟨version⟩}%
        {⟨author⟩}{⟨email⟩}{⟨address⟩}
⟨body of the option⟩
\xyendinput
```

The 6 arguments to `\xyprovide` should contain the following:

⟨option⟩ Option load name as used in the `\xyoption` command. This should be safe and distinguishable for any operating system and is thus limited to characters chosen among the lowercase letters (`a`–`z`), digits (`0`–`9`), and dash (`-`), and all options should be uniquely identifiable by the first 6 (six) characters only.

⟨name⟩ Descriptive name for the option.

⟨version⟩ Identification of the version of the option.

⟨author⟩ The name(s) of the author(s).

⟨email⟩ The electronic mail address(es) of the author(s) *or* the affiliation if no email is available.

⟨address⟩ The postal address(es) of the author(s).

This information is used not only to print a nice banner but also to (1) silently skip loading if the same version was preloaded and (2) print an error message if a different version was preloaded.

The 'dummy' option described in §22 is a minimal option using the above features. It uses the special `DOCMODE` format to include its own documentation for this document (like all official XY-pic options) but this is not a requirement.

7.3 Driver options

The ⟨driver⟩ options described in part IV require special attention because each driver can support several extension options, and it is sometimes desirable to change ⟨driver⟩ or even mix the support provided by several.[7]

A ⟨driver⟩ option is loaded as other options with `\xyoption{`⟨driver⟩`}` (or through LaTeX 2_ε class or package options as described in §1.1). The special thing about a ⟨driver⟩ is that loading it simply declares the name of it, establishes what extensions it will support, and selects it temporarily. Thus the special capabilities of the driver will only be exploited in the produced DVI file if some of these extensions are also loaded and if the driver is still selected when output is produced. Generally, the order in which the options are loaded is immaterial. (Known exceptions affect only internal processing and are not visible to the user in terms of language and expected output.) In particular one driver can be preloaded in a format and a different one used for a particular document.

The following declarations control this:

`\UseSingleDriver`	forces one driver only
`\MultipleDrivers`	allows multiple drivers
`\xyReloadDrivers`	resets driver information

The first command restores the default behaviour: that ony one ⟨driver⟩ is allowed, *i.e.*, each loading of a ⟨driver⟩ option cancels the previous. The second allows consecutive loading of drivers such that when loading a ⟨driver⟩ only the extensions actually supported are selected, leaving other extensions supported by previously selected drivers untouched. Beware that this can be used to create DVI files that cannot be processed by any actual DVI driver program!

The last command is sometimes required to reset the XY-pic ⟨driver⟩ information to a sane state, for example, after having applied one of the other two in the middle of a document, or when using simple formats like plain TeX that do not have a clearly distinguished preamble.

As the above suggests it sometimes makes sense to load ⟨driver⟩s in the actual textual part of a document, however, it is recommended that only drivers *also* loaded in the preamble are reloaded later, and that `\xyReloadDrivers` is used when there is doubt about the state of affairs. In case of confusion the special command `\xyShowDrivers` will list all the presently supported and selected driver-extension pairs to the TeX log.

It is not difficult to add support for additional ⟨driver⟩s; how is described in the TeXnical documentation.

Most extensions will print a warning when a capability is used which is not supported by the presently loaded ⟨driver⟩. Such messages are only printed once, however, (for some formats they are repeated at the end). Similarly, when the support of an extension that exploits a particular ⟨driver⟩ is used a warning message will be issued that the DVI file is not portable.

Part II
Extensions

This part documents the graphic capabilities added by each standard extension option. For each is indicated the described version number, the author, and how it is loaded.

Many of these are only fully supported when a suitable *driver* option (described in part IV) is also loaded, however, all added constructions are always *accepted* even when not supported.

8 Curve and Spline extension

Vers. 3.7 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{curve}`

This option provides XY-pic with the ability to typeset spline curves by constructing curved connections using arbitrary directional objects and by encircling objects similarly. *Warning*: Using curves can be quite a strain on TeX's memory; you should therefore limit the length and number of curves used on a single page. Memory use is less when combined with a backend capable of producing its own curves; *e.g.*, the PostScript backend).

8.1 Curved connections

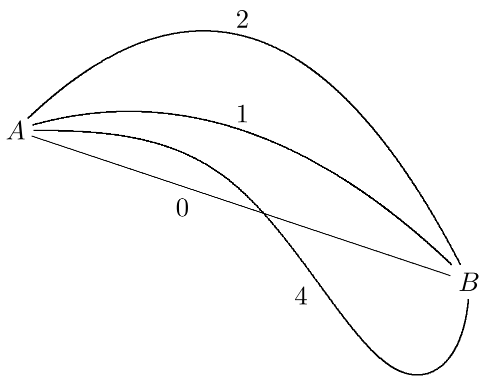
Simple ways to specify curves in XY-pic are as follows:

`**\crv{`⟨poslist⟩`}` curved connection

[7] The kernel support described here is based on the (now defunct) `xydriver` include file by Ross Moore.

`**\crvs{`⟨dir⟩`}`	get ⟨poslist⟩ from the stack
`\curve{`⟨poslist⟩`}`	as a ⟨decor⟩ation

in which ⟨poslist⟩ is a list of valid ⟨pos⟩itions. The decoration form `\curve` is just an abbreviation for `\connect\crv`. As usual, the current p and c are used as the start and finish of the connection, respectively. Within ⟨poslist⟩ the ⟨pos⟩itions are separated by `&`. A full description of the syntax for `\crv` is given in figure 7.



If ⟨poslist⟩ is empty a straight connection is computed. When the length of ⟨poslist⟩ is one or two then the curve is uniquely determined as a single-segment Bézier quadratic or cubic spline. The tangents at p and c are along the lines connecting with the adjacent control point. With three or more ⟨pos⟩itions a cubic B-spline construction is used. Bézier cubic segments are calculated from the given control points.

The previous picture was typeset using:

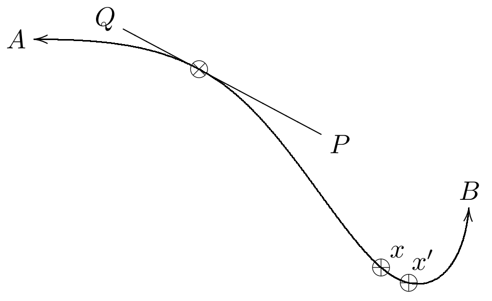
```
\xy (0,20)*+{A};(60,0)*+{B}
**\crv{}
**\crv{(30,30)}
**\crv{(20,40)&(40,40)}
**\crv{(10,20)&(30,20)&(50,-20)&(60,-10)}
\endxy
```

except for the labels, which denote the number of entries in the ⟨poslist⟩. (Extending this code to include the labels is set below as an exercise).

The `?`-operator of §3 (note 3h) is used to find arbitrary ⟨place⟩s along a curve in the usual way.

Exercise 16: Extend the code given for the curves in the previous picture so as to add the labels giving the number of control points.

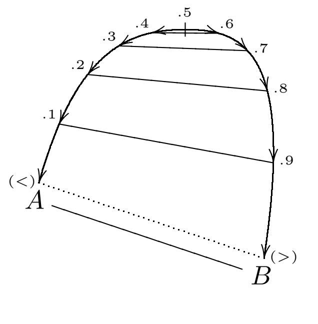
Using `?` will set the current direction to be tangential at that ⟨place⟩, and one can ⟨slide⟩ specified distances along the curve from a found ⟨place⟩ using the `?`.../⟨dimen⟩`/` notation:



Exercise 17: Suggest code to produce something like the above picture; the spline curve is the same as in the previous picture. *Hints*: The line is 140pt long and touches 0.28 of the way from A to B and the x is 0.65 of the way from A to B.

The positions in ⟨poslist⟩ specify *control points* which determine the initial and final directions of the curve—leaving p and arriving at c—and how the curve behaves in between, using standard spline constructions. In general, control points need not lie upon the actual curve.

A natural spline parameter varies in the interval $[0, 1]$ monotonically along the curve from p to c. This is used to specify ⟨place⟩s along the curve, however there is no easy relation to arc-length. Generally the parameter varies more rapidly where the curvature is greatest. The following diagram illustrates this effect for a cubic spline of two segments (3 control points).



Exercise 18: Write code to produce a picture such as the one above. (*Hint*: Save the locations of places along the curve for later use with straight connections.)

To have the same ⟨pos⟩ occuring as a multiple control point simply use a delimiter, which leaves the ⟨pos⟩ unchanged. Thus `\curve{`⟨pos⟩`&}` uses a cubic spline, whereas `\curve{`⟨pos⟩`}` is quadratic.

Repeating the same control point three times in succession results in straight segments to that control point. Using the default styles this is an expensive way to get straight lines, but it allows for extra effects with other styles.

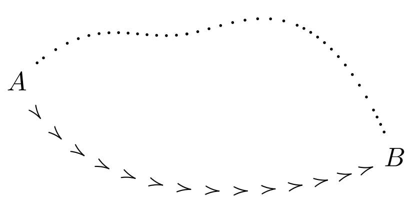
Syntax			Action
`\curve`⟨modifier⟩`{`⟨curve-object⟩⟨poslist⟩`}`			construct curved connection
⟨modifier⟩	$\longrightarrow$	⟨empty⟩	zero or more modifiers possible; default is `~C`
	|	`~`⟨curve-option⟩ ⟨modifier⟩	set ⟨curve-option⟩
⟨curve-option⟩	$\longrightarrow$	`p` | `P` | `l` | `L` | `c` | `C`	show only[8d] control points (`p`=points), joined by lines (`l`=lines), or curve only (`c`=curve)
	|	`pc` | `pC` | `Pc` | `PC`	show control points[8f] and curve[8e]
	|	`lc` | `lC` | `Lc` | `LC`	show lines joining[8g] control points and curve[8e]
	|	`cC`	plot curve twice, with and without specified formatting
⟨curve-object⟩	$\longrightarrow$	⟨empty⟩	use the appropriate default style
	|	`~*`⟨object⟩ ⟨curve-object⟩	specify the “drop” object[8a] and maybe more[8c]
	|	`~**`⟨object⟩ ⟨curve-object⟩	specify “connect” object[8b] and maybe more[8c]
⟨poslist⟩	$\longrightarrow$	⟨empty⟩ | ⟨pos⟩ ⟨delim⟩ ⟨poslist⟩	list of positions for control points
	|	`~@` | `~@` ⟨delim⟩ ⟨poslist⟩	add the current stack[8h] to the control points
⟨delim⟩	$\longrightarrow$	`&`	allowable delimiter

Figure 7: Syntax for curves.

Notes

8a. The “drop” object is set once, then “dropped” many times at appropriately spaced places along the curve. If directional, the direction from p to c is used. Default behaviour is to have tiny dots spaced sufficiently closely as to give the appearance of a smooth curve. Specifying a larger size for the “drop” object is a way of getting a dotted curve (see the example in the next note).

8b. The “connect” object is also dropped at each place along the curve. However, if non-empty, this object uses the tangent direction at each place. This allows a directional object to be specified, whose orientation will always match the tangent. To adjust the spacing of such objects, use an empty “drop” object of non-zero size as shown here:



```
\xy (0,0)*+{A}; (50,-10)*+{B}
**\crv{~*=<4pt>{.} (10,10)&(20,0)&(40,15)}
**\crv{~*=<8pt>{}~**!/-5pt/\dir{>}(10,-20)
 &(40,-15)} \endxy
```

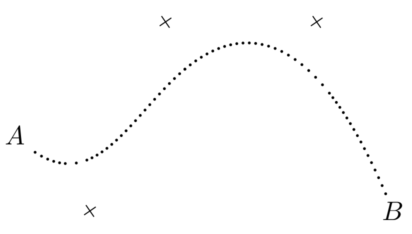
When there is no “connect” object then the tangent calculations are not carried out, resulting in a saving of time and memory; this is the default behaviour.

8c. The “drop” and “connect” objects can be specified as many times as desired. Only the last specification of each type will actually have any effect. (This makes it easy to experiment with different styles.)

8d. Complicated diagrams having several spline curves can take quite a long time to process and may use a lot of TEX’s memory. A convenient device, especially while developing a picture, is to show only the location of the control points or to join the control points with lines, as a stylized approximation to the spline curve. The ⟨curve-option⟩s `~p` and `~l` are provided for this purpose. Uppercase versions `~P` and `~L` do the same thing but use any ⟨curve-object⟩s that may be specified, whereas the lowercase versions use plain defaults: small cross for `~p`, straight line for `~l`. Similarly `~C` and `~c` set the spline curve using any specified ⟨curve-option⟩s or as a (default) plain curve.

8e. Use of `~p`, `~l`, etc. is extended to enable both the curve and the control points to be easily shown in the same picture. Mixing upper- and lower-case specifies whether the ⟨curve-option⟩s are to be applied to the spline curve or the (lines joining) control points. See the examples accompanying the next two notes.

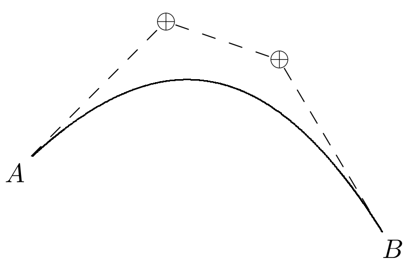
8f. By default the control points are marked with a small cross, specified by `*\dir{x}`. The "connect" object is ignored completely.



was typeset by ...

```
\xy (0,0)*+{A};(50,-10)*+{B}
**\crv~pC{~*=<\jot>{.}(10,-10)&(20,15)
 &(40,15)} \endxy
```

8g. With lines connecting control points the default "drop" object is empty, while the "connect" object is `\dir{-}` for simple straight lines. If non-empty, the "drop" object is placed at each control point. The "connect" object may be used to specify a fancy line style.

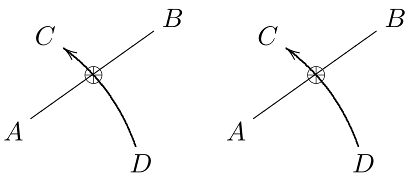


was typeset by ...

```
\xy (0,0)*+{A};(50,-10)*+{B}
**\crv~Lc{~**\dir{--}~*{\oplus}
 (20,20)&(35,15)} \endxy
```

8h. When a stack of ⟨pos⟩itions has been established using the `@i` and `@+` commands, these positions can be used and are appended to the ⟨poslist⟩.

Intersection with a curved connection Just as the intersection of two lines (3j) can be found, so can the intersection of a straight line with a curved connection, or the intersection of a curve with a straight connection.



```
\xy*+{A}="A";p+/r5pc/+(0,15)*+{B}="B"
 ,p+<1pc,3pc>*+{C}="C"
 ,"A"+<4pc,-1pc>*+{D}="D",{\ar@/_/"C"}
 ,?!{"A";"B"**@{-}}*++{\oplus}
\endxy \quad \xy
 *+{A}="A";p+/r5pc/+(0,15)*+{B}="B",
 ,p+<1pc,3pc>*+{C}="C"
 ,"A"+<4pc,-1pc>*+{D}="D","A";"B"**@{-}
 ,?!{"D",{\ar@/_/"C"}}*++{\oplus}
\endxy
```

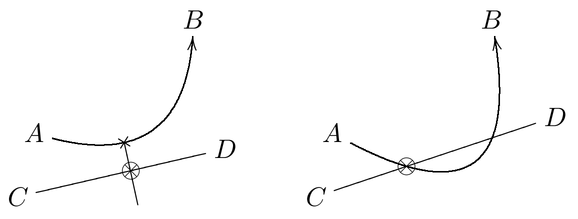
When the line separates the end-points of a curve an intersection can always be found. If there is more than one then that occurring earliest along the curve is the one found.

If the line does not separate the end-points then there may be no intersection with the curve. If there is one then either the line is tangential or necessarily there will also be at least one other intersection. A message

```
perhaps no curve intersection, or many.
```

is written to the log-file, but a search for an intersection will still be performed and a "sensible" place found on the curve. In the usual case of a single quadratic or cubic segment, the place nearest the line is found and the tangent direction is established.

The following examples show this, and show how to get the place on the line nearest to the curve.



```
\xy *+{A}="A";p+/r5pc/+(0,15)*+{B}="B",
 ,p-<.5pc,2pc>*+{C}="C","A"+<6pc,-.5pc>
 ,*+{D}="D","A",{\ar@/_25pt/"B"}
 ,?!{"C";"D"**@{-}}*\dir{x}="E"
 ,+/_2pc/="F";"E"**@{-},?!{"C";"D"}
 ,*{\otimes}\endxy\qquad\xy
 *+{A}="A";p+/r5pc/+(0,15)*+{B}="B",
 ,p-<.5pc,2pc>*+{C}="C"
 ,"A"+<7pc,.5pc>*+{D}="D","A"
 ,{\ar@/_40pt/"B"},?!{"C";"D"**@{-}}
 ,*{\otimes}\endxy
```

Sometimes TEX will run short of memory when many curves are used without a backend with special support for curves. In that case the following commands, that obey normal TEX groupings, may be helpful:

```
\SloppyCurves
\splinetolerance{⟨dimen⟩}
```

allow adjustment of the tolerance used to typeset curves. The first sets tolerance to .8pt, after which

`\splinetolerance{0pt}` resets to the original default of fine curves.

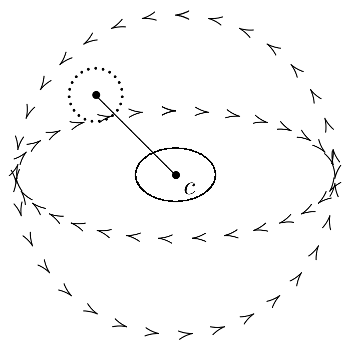
8.2 Circles and Ellipses

Here we describe the means to a specify circles of arbitrary radius, drawn with arbitrary line styles. When large-sized objects are used they are regularly spaced around the circle. Similarly ellipses may be specified, but only those having major/minor axes aligned in the standard directions; spacing of objects is no longer regular, but is bunched toward the narrower ends.

Such a circle or ellipse is specified using...

`\xycircle`⟨vector⟩`{`⟨style⟩`}`

where the components of the ⟨vector⟩ determine the lengths of the axis for the ellipse; thus giving a circle when equal. The ⟨style⟩ can be any ⟨conn⟩, as in 14 that works with curved arrows—many do. Alternatively ⟨style⟩ can be any ⟨object⟩, which will be placed equally-spaced about the circle at a separation to snugly fit the ⟨object⟩s. If ⟨empty⟩ then a solid circle or ellipse is drawn.



```
\xy 0;/r5pc/:*\dir{*}
 ;p+(.5,-.5)*\dir{*}="c"
,**\dir{-},*+!UL{c},"c"
,*\xycircle(1,.4){++\dir{<}}
,*\xycircle(1,1){++\dir{>}}
,*\xycircle<15pt,10pt>{}
;*\xycircle<10pt>{{.}}
\endxy
```

8.3 Quadratic Splines

Quadratic Bézier splines, as distinct from cubic Bézier splines, are constructed from parabolic arcs, using 'control points' to determine the tangents where successive arcs are joined.

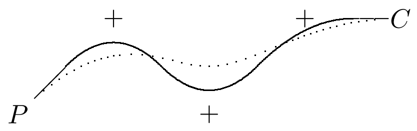
Various implementations of such curves exist. The one adopted here is consistent with the `xfig` drawing utility and TPIC implementations. These have the property of beginning and ending with straight segments, half the length to the corresponding adjacent control-point. Furthermore the midpoint between successive control-points lies on the spline, with the line joining the control-points being tangent there.

Such curves are specified, either as a ⟨decor⟩ or as an ⟨object⟩, using...

`\qspline{`⟨style⟩`}`

where the start and end of the curve are at p and c respectively. The control-points are taken from the current stack, see 3o. If this stack is empty then a straight line is constructed.

The following example compares the quadratic spline with the gentler curving B-spline having the same control points, using `\crvs`.



```
\xy /r1.5pc/:,+<5pc,3pc>*+{P};p
@(,+(2,2)*{+}@+, +(2,-2)*{+}@+
,+(2,2)*{+}@+, +(2,0)*+{C}="C"
,*\qspline{},"C",**\crvs{.}
,@i @)\endxy
```

9 Frame and Bracket extension

Vers. 3.7 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩
Load as: `\xyoption{frame}`

The `frame` extension provides a variety of ways to puts frames in XY-pictures.

The frames are XY-pic ⟨object⟩s on the form

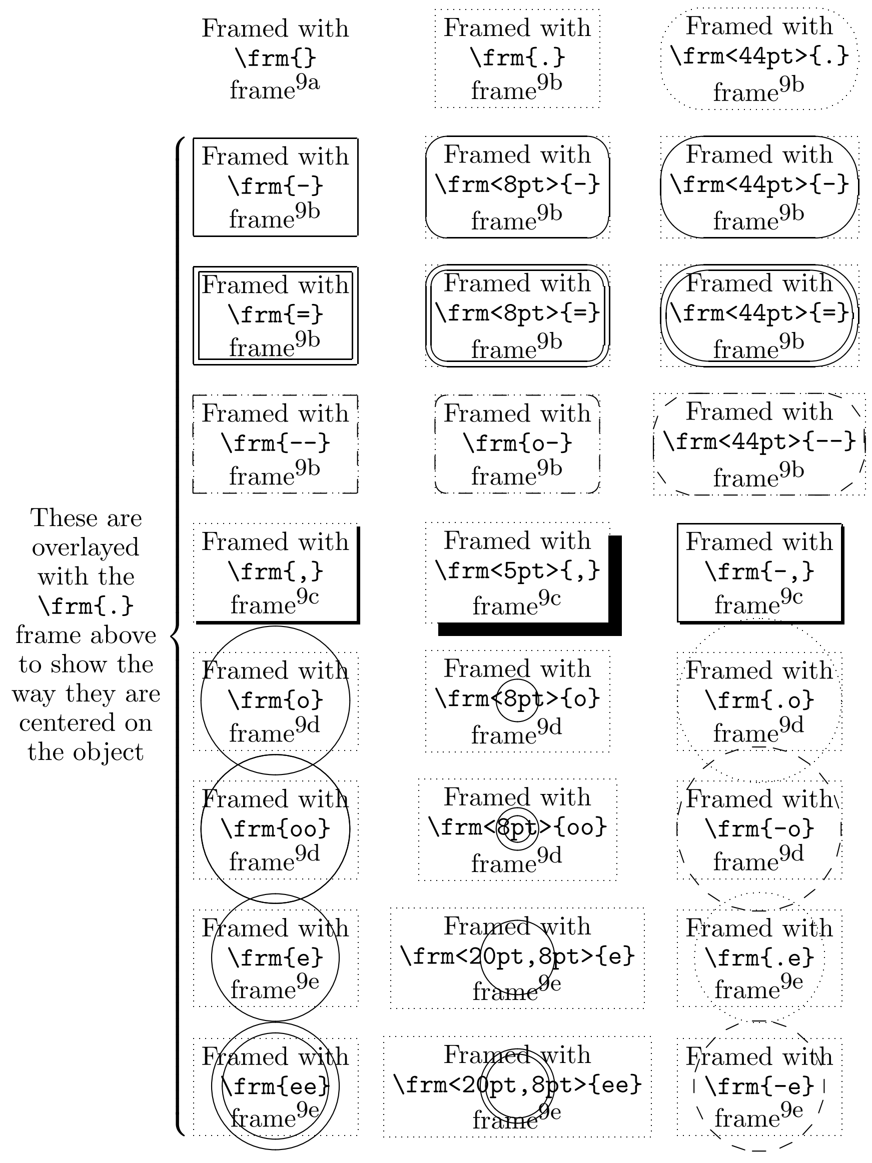
`\frm{` ⟨frame⟩ `}`

to be used in ⟨pos⟩itions: Dropping a frame with `*`...`\frm{`⟨frame⟩`}` will frame the c object; connecting with `**`...`\frm{`...⟨frame⟩`}` will frame the result of $c.p$.

Below we distinguish between 'ordinary' frames, 'brackets' and 'fills'; last we present how some frames can be added to other objects using object modifier ⟨shape⟩s.

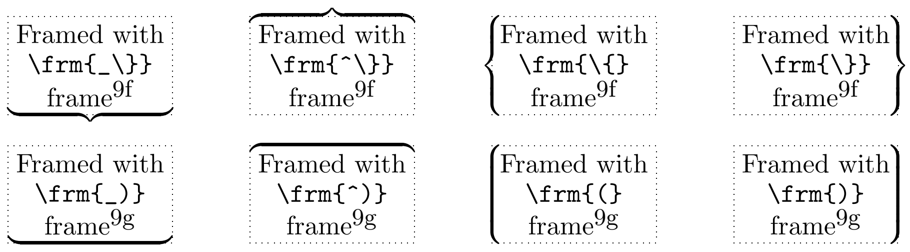
9.1 Frames

Figure 8 shows the possible frames and the applicable ⟨modifier⟩s with reference to the notes below.



Framed with \frm{} frame[9a]
Framed with \frm{.} frame[9b]
Framed with \frm<44pt>{.} frame[9b]
Framed with \frm{-} frame[9b]
Framed with \frm<8pt>{-} frame[9b]
Framed with \frm<44pt>{-} frame[9b]
Framed with \frm{=} frame[9b]
Framed with \frm<8pt>{=} frame[9b]
Framed with \frm<44pt>{=} frame[9b]
Framed with \frm{--} frame[9b]
Framed with \frm{o-} frame[9b]
Framed with \frm<44pt>{--} frame[9b]
These are overlayed with the \frm{.} frame above to show the way they are centered on the object
Framed with \frm{,} frame[9c]
Framed with \frm<5pt>{,} frame[9c]
Framed with \frm{-,} frame[9c]
Framed with \frm{o} frame[9d]
Framed with \frm<8pt>{o} frame[9d]
Framed with \frm{.o} frame[9d]
Framed with \frm{oo} frame[9d]
Framed with \frm<8pt>{oo} frame[9d]
Framed with \frm{-o} frame[9d]
Framed with \frm{e} frame[9e]
Framed with \frm<20pt,8pt>{e} frame[9e]
Framed with \frm{.e} frame[9e]
Framed with \frm{ee} frame[9e]
Framed with \frm<20pt,8pt>{ee} frame[9e]
Framed with \frm{-e} frame[9e]

Figure 8: Plain ⟨frame⟩s.



Framed with \frm{_\}} frame[9f]
Framed with \frm{^\}} frame[9f]
Framed with \frm{\{} frame[9f]
Framed with \frm{\}} frame[9f]
Framed with \frm{_)} frame[9g]
Framed with \frm{^)} frame[9g]
Framed with \frm{(} frame[9g]
Framed with \frm{)} frame[9g]

Figure 9: Bracket ⟨frame⟩s.

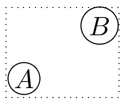
Notes

9a. The `\frm{}` frame is a dummy useful for not putting a frame on something, *e.g.*, in macros that take a ⟨frame⟩ argument.

9b. *Rectangular* frames include `\frm{.}`, `\frm{-}`, `\frm{=}`, `\frm{--}`, `\frm{==}`, and `\frm{o-}`. They all make rectangular frames that essentially trace the border of a rectangle-shaped object.

The ⟨frame⟩s `\frm{-}` and `\frm{=}` allow an optional *corner radius* that rounds the corners of the frame with quarter circles of the specified radius. This is not allowed for the other frames—the `\frm{o-}` frame always gives rounded corners of the same size as the used dashes (when `\xydashfont` is the default one then these are `5pt` in radius).

Exercise 19: How do you think the author typeset the following?



9c. The frame `\frm{,}` puts a shade, built from rules, into the picture beneath the (assumed rectangular) object, thereby giving the illusion of 'lifting' it; `\frm<`⟨dimen⟩`>{,}` makes this shade ⟨dimen⟩ deep.

`\frm{-,}` combines a `\frm{-}` with a `\frm{,}`.

9d. Circles done with `\frm{o}` have radius as $(R + L)/2$ and with `\frm<`⟨dimen⟩`>{o}` have radius as the ⟨dimen⟩; `\frm{oo}` makes a double circle with the outermost circle being the same as that of `\frm{o}`.

Exercise 20: What is the difference between `*\cir{}` and `*\frm{o}`?

9e. Ellipses specified using `\frm{e}` have axis lengths $(R + L)/2$ and $(U + D)/2$, while those with `\frm<`⟨dimen,dimen⟩`>{e}` use the given lengths for the axes. `\frm{ee}` makes a double ellipse with outermost ellipse being the same as that of `\frm{e}`.

Without special support to render the ellipses, either via a ⟨driver⟩ or using the `arc` feature, the ellipse will be drawn as a circle of radius approximately the average of the major and minor axes.

To Do: Allow ⟨frame variant⟩s like those used for directionals, *i.e.*, `\frm2{-}` should be the same as `\frm{=}`. Add `\frm{o,}` and more brackets.

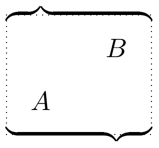
9.2 Brackets

The possible brackets are shown in figure 9 with notes below.

Notes

9f. *Braces* are just the standard plain TeX large braces inserted correctly in XY-pic pictures with the 'nib' aligned with the reference point of the object they brace.

Exercise 21: How do you think the author typeset the following?

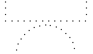
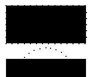
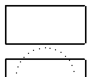
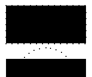
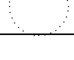
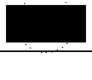
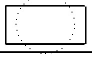
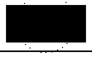


9g. *Parenthesis* are like braces except they have no nib and thus do not depend on where the reference point of c is.

Bug: The brackets above require that the computer modern `cmex` font is loaded in TeX font position 3.

9.3 Filled regions

In addition to the above there is a special frame that "fills" the inside of the current object with ink: `\frm {*}` and `\frm {**}`; the latter is intended for *emphasizing* and thus "strokes" the outline, using the thinnest black line available on the printer or output device; furthermore it moits the actual filling in case this would obscure further text typeset on top. Some alteration to the shape is possible, using `*\frm<dimen>{*}`. Hence rectangular, oval, circular and elliptical shapes can be specified for filling. The following examples illustrate this in each case:

⟨object⟩	`\frm{*}`	`\frm{**}`	`\frm<6pt>{*}`
			
			

However, filling non-rectangular shapes will result in a rectangle unless a driver is used that supports arbitrary filling. With some drivers the above fills will thus all be identical, as rectangular.

9.4 Framing as object modifier

In addition, frames may be accessed using the special `[F`⟨frame⟩`]` object modifier ⟨shape⟩s that will add the desired ⟨frame⟩ to the current object. The frame appropriate to the edge of the object will be chosen (presently either rectangular or elliptical).

If shape modifiers need to be applied to the ⟨frame⟩ alone then they can be included using `:` as separator. Thus `[F-:red]` will make a red frame (provided the color extension is active, of course). Additionally the variant of frames using `<`⟨dimen⟩`>` can be accessed by specifying `[`. . . `:<`⟨dimen⟩`>]`.

Here are some simple examples using this feature.

$\boxed{\boxed{\text{text with background}}}$

$\boxed{\textbf{bold white on black}}$

```
\xy *+<1.5pt>[F**:white]++[F**:red]
\txt{text with background}
,+!D+/d1pc/,*++[F**:black][white]
\txt\bf{bold white on black}\endxy
```

Notice that when multiple frame-modifiers are used, the frames are actually placed in reverse order, so that earlier ones are printed on top of later ones.

To Do: The frame option is not quite complete yet: some new frames and several new brackets should be added.

9.5 Using curves for frames

If the curve option is loaded, then circular and elliptical frames of arbitrary radius can be constructed, by specifying `\UseCurvedFrames`. This can be negated by `\UseFontFrames`. Both of these commands obey normal TEX grouping. Furthermore, dotted and dashed frames now have a regular spacing of their constituent objects. The usual warnings about memory requirements for large numbers of curves apply here also.

10 More Tips extension

Vers. 3.3 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{tips}`

This extension provides several additional styles of 'tips' for use (primarily) as arrow heads, and makes it possible to define customised tips. This is used to support tips that mimic the style of the Computer Modern fonts[8] by Knuth (see [7] and [6, appendix F]) and of the Euler math fonts distributed by the $\mathcal{AMS}$.

Font selection is done with the command

`\SelectTips {`⟨family⟩`} {`⟨size⟩`}`

where the ⟨family⟩ and ⟨size⟩ should be selected from the following table.

Family	10	11	12
`xy`	→ ⇒ ⇛	→ ⇒ ⇛	→ ⇒ ⇛
`cm`	→ ⇒ ⇛	→ ⇒ ⇛	→ ⇒ ⇛
`eu`	→ ⇒ ⇛	→ ⇒ ⇛	→ ⇒ ⇛

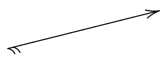
Once a selection is made, the following commands are available:

`\UseTips`	activate selected tips
`\NoTips`	deactivate

They are local and thus can be switched on and/or off for individual pictures using the TEX grouping mechanism, *e.g.*,

```
\SelectTips{cm}{10}
\xy*{} \ar
 @{*{\UseTips\dir_{<<}}-*{\NoTips\dir{>}}}
 (20,5)*{} \endxy
```

will typeset



regardless of which tips are used otherwise in the document.

11 Line styles extension

Vers. 3.6 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{line}`

This extension provides the ability to request various effects related to the appearance of straight lines; *e.g.*. thickness, non-standard dashing, and colour.

These are effects which are not normally available within TEX. Instead they require a suitable 'back-end' option to provide the necessary `\special` commands, or extra fonts, together with appropriate commands to implement the effects. Thus

$\boxed{\begin{array}{c}\text{Using this extension will have no}\\ \text{effect on the output unless used with}\\ \text{a backend that explicitly supports it.}\end{array}}$

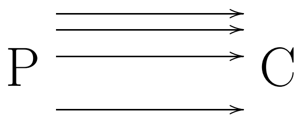
The extension provides special effects that can be used with any XY-pic ⟨object⟩, by defining `[`⟨shape⟩`]` modifiers. The modification is local to the ⟨object⟩ currently being built, so will have no effect if this object is never actually used.

[8]This function was earlier supported by the `cmtip` extension which is still included in the distribution but is now obsolete.

Adjusting line thickness The following table lists the modifiers primarily to alter the thickness of lines used by XY-pic. They come in two types — either a single keyword, or using the key-character `|` with the following text parsed.

`[thicker]`	double line thickness
`[thinner]`	halve line thickness
`[|(`⟨num⟩`)]`	multiple of usual thickness
`[|<`⟨dimen⟩`>]`	set thickness to ⟨dimen⟩
`[|`⟨dimen⟩`]`	also sets to ⟨dimen⟩
`[|=`⟨word⟩`]`	make `[`⟨word⟩`]` set current style settings
`[|*]`	reuse previous style
`[butt]`	butt cap at ends
`[roundcap]`	round cap at ends
`[projcap]`	projecting square cap.

Later settings of the linewidth override earlier settings; multiple calls to `[thicker]` and `[thinner]` compound, but the other variants set an absolute thickness. The line-thickness specification affects arrow-tips as well as the thickness of straight lines and curves. Three kinds of line-caps are available; they are discussed below in the section on 'poly-lines'.



```
\xy/r8pc/:*++\txt\huge{C}="c"
,0*++\txt\huge{P}="p",
,"p",{\ar@*{[|(1)]}"p";"c"<20pt>}
,"p",{\ar@*{[|(4)]}"p";"c"<14pt>}
,"p",{\ar@*{[|(10)]}"p";"c"<4pt>}
,"p",{\ar@*{[|(20)]}"p";"c"<-16pt>}
\endxy
```

Using the PostScript back-end, the size of the arrow-head grows aesthetically with the thickness of the line used to draw it. This growth varies as the square-root of the thickness; thus for very thick lines (20+ times normal) the arrowhead begins to merge with the stem.

The diagram in figure 10, page 31, uses different line-thicknesses and colours.

Poly-lines By a 'poly-line' we mean a path built from straight line segments having no gaps where each segment abuts the next. The poly-line could be the edges of a polygon, either closed or open if the end-points are different.

The reason for considering a poly-line as a separate ⟨object⟩, rather than simply as a ⟨path⟩ built from straight lines, becomes apparent only when the lines have appreciable thickness. Then there are several standard ways to fashion the 'joins' (where segments meet). Also the shape of the 'caps' at either end of the poly-line can be altered.

The following modifiers are used to determine the shapes of the line 'caps' and 'joins':

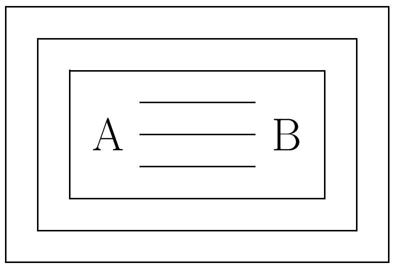
`[|J`⟨val⟩`]`	join style, ⟨val⟩ = 0, 1 or 2
`[mitre]`	mitre-join, same as `[|J0]`
`[roundjoin]`	round join, same as `[|J1]`
`[bevel]`	bevel-join, same as `[|J2]`
`[|C`⟨val⟩`]`	end-cap, ⟨val⟩ = 0, 1 or 2
`[butt]`	"butt" cap, same as `[|C0]`
`[roundcap]`	round cap, same as `[|C1]`
`[projcap]`	"projecting square" cap, same as `[|C2]`
`[|M(`⟨num⟩`)]`	set mitrelimit to ⟨num⟩≥ 1

These effects are currently implemented only with the PostScript back-end or when using `\xypolyline` (described below) with a PostScript ⟨driver⟩. In this case the 'cap' setting can be applied to any segment, straight or curved, whether part of a poly-line or not; however the 'join' setting applies only to poly-lines. Arrow-tips are not affected. The defaults are to use round joins and round-cap ends.

Adjusting the miter-limit affects how far miters are allowed to protrude when two wide lines meet at small angles. The ⟨num⟩ is in units of the line-thickness. Higher values mean using bevels only at smaller angles, while the value of 1 is equivalent to using bevels at all angles. The default miter-limit is 10.

The path taken by the 'poly-line' this is read as the list of ⟨pos⟩itions in the current 'stack', ignoring size extents. The macro `\xypolyline` is used as a ⟨decor⟩; it reads the ⟨pos⟩itions from the stack, but leaves the stack intact for later use.

The following diagram illustrates the use of line-thickness, line-joins and line-caps with poly-lines. It contains an example of each of the styles.



```
\xycompileto{poly}%
{/r4pc/:,*[|<5pt>][thicker]\xybox{%
 *+(3,2){}="X"
;@={p+CU,p+LU,p+LD,p+RD,p+RU,p+CU}
 ,{0*[miter]\xypolyline{}}
 ,{\xypolyline{*}},@i@)
,"X",*+(2.5,1.5){}="X"
,@={!CU,!LU,!LD,!RD,!RU,!CU}
 ,{0*[gray][roundjoin]\xypolyline{}}
 ,{0*[gray]\xypolyline{*}},@i@)
,"X",*+(2,1){}="X"
,@={!CU,!LU,!LD,!RD,!RU,!CU}
 ,{0*[white]\xypolyline{*}}
 ,{0*[bevel]\xypolyline{}},@i@)
,"X"-(.7,0)*++\txt\LARGE{A}="a"
,"X"+(.7,0)*++\txt\LARGE{B}="b"
,{\ar@{-}@*{[butt][thinner]}"a";"b"<1pc>}
,{\ar@{-}@*{[roundcap][thinner]}"a";"b"}
,{\ar@{-}@*{[projcap][thinner]}"a";"b"<-1pc>}
}}
```

Note the use of `{0*[...]\xypolyline{..}}` to apply style-modifiers to a polyline. The `@={!..}` method for loading the stack gives equivalent results to using `;@={p+..}`, since `\xypolyline` ignores the edge extents of each ⟨pos⟩ in the stack.

Note also that the argument `#1` to `\xypolyline` affects what is typeset. Allowable arguments are:

`\xypolyline{}`	solid line
`\xypolyline{.}`	dotted line
`\xypolyline{-}`	dashed line
`\xypolyline{*}`	fill enclosed polygon
`\xypolyline{?}`	fill enclosed polygon using even-odd rule
`\xypolyline{{*}}`	use `\dir{*}` for lines
`\xypolyline{<toks>}`	using `\dir{<toks>}`

The latter cases one has `**\dir{...}` being used to connect the vertices of the polyline, with `{{*}}` being needed to get `**\dir{*}`. Similarly `**\dir` is used when a ⟨driver⟩ is not available to specifically support polylines; in particular the two 'fill' options `*` and `?` will result in a dotted polygon outline the region intended to be filled.

In all cases it is up to the user to load the stack before calling `\xypolyline{...}`. A particularly common case is the outline of an existing XY-pic ⟨object⟩, as in the example above. Future extensions to `\frm` will provide a simplified mechanism whereby the user need not call `\xypolyline` explicitly for such effects.

12 Rotate and Scale extension

Vers. 3.3 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{rotate}`

This extension provides the ability to request that any object be displayed rotated at any angle as well as scaled in various ways.

These are effects which are not normally available within TEX. Instead they require a suitable 'back-end' option to provide the necessary `\special` commands, or extra fonts, together with appropriate commands to implement the effects. Thus

Using this extension will have no effect on the output unless used with a backend that explicitly supports it.

The extension provides special effects that can be used with any XY-pic ⟨object⟩ by defining `[`⟨shape⟩`]` modifiers. The modification is local to the ⟨object⟩ currently being built, so will have no effect if this object is never actually used.

The following table lists the modifiers that have so far been defined. They come in two types – either a single keyword, or a key-character with the following text treated as a single argument.

`[@]`	align with current direction
`[@`⟨direction⟩`]`	align to ⟨direction⟩
`[@!`⟨number⟩`]`	rotate ⟨number⟩ degrees
`[*`⟨number⟩`]`	scale by ⟨number⟩
`[*`$\langle\text{num}\rangle_x$`,`$\langle\text{num}\rangle_y$`]`	scale x and y separately
`[left]`	rotate anticlockwise by 90°
`[right]`	rotate (clockwise) by 90°
`[flip]`	rotate by 180°; same as `[*-1,-1]`
`[dblsize]`	scale to double size
`[halfsize]`	scale to half size

These `[`⟨shape⟩`]` modifiers specify transformations of the ⟨object⟩ currently being built. If the object has a rectangle edge then the size of the rectangle is transformed to enclose the transformed object; with a circle edge the radius is altered appropriately.

Each successive transformation acts upon the result of all previous. One consequence of this is that the order of the shape modifiers can make a significant difference in appearance—in general, transformations do not commute. Even successive rotations can give different sized rectangles if taken in the reverse order.

Sometimes this change of size is not desirable. The following commands are provided to modify this behaviour.

`\NoResizing`	prevents size adjustment
`\UseResizing`	restores size adjustments

The `\NoResizing` command is also useful to have at the beginning of a document being typeset using a driver that cannot support scaling effects, in particular when applied to whole diagrams. In any case an unscaled version will result, but now the spacing and positioning will be appropriate to the unscaled rather than the scaled size.

Scaling and Scaled Text The ⟨shape⟩ modifier can contain either a single scale factor, or a pair indicating different factors in the x- and y-directions. Negative values are allowed, to obtain reflections in the coordinate axes, but not zero.

Rotation and Rotated Text Within `[@...]` the `...` are parsed as a ⟨direction⟩ locally, based on the current direction. The value of count register `\Direction` contains the information to determine the requested direction. When no ⟨direction⟩ is parsed then `[@]` requests a rotation to align with the current direction.

The special sequence `[@!...]` is provided to pass an angle directly to the back-end. The XY-pic size and shape of the ⟨object⟩ with `\rectangleEdge` is unchanged, even though the printed form may appear rotated. This is a feature that must be implemented specially by the back-end. For example, using the PostScript back-end, `[@!45]` will show the object rotated by 45° inside a box of the size of the unrotated object.

To Do: Provide example of repeated, named transformation.

Reflections Reflections can be specified by a combination of rotation and a flip — either `[hflip]` or `[vflip]`.

Shear transformations **To Do:** Provide the structure to support these; then implement it in PostScript.

Example The diagram in figure 10 illustrates many of the effects described above as well as some additional ones defined by the `color` and `rotate` extensions.

Exercise 22: Suggest the code used by the author to typeset 10.

The actual code is given in the solution to the exercise. Use it as a test of the capabilities of your DVI-driver. The labels should fit snugly inside the accompanying rectangles, rotated and flipped appropriately.

Bug: This figure also uses colours, alters line-thickness and includes some PostScript drawing. The colours may print as shades of gray, with the line from A to B being thicker than normal. The wider band sloping downwards may have different width and length according to the DVI-driver used; this depends on the coordinate system used by the driver, when 'raw' PostScript code is included.

13 Colour extension

Vers. 3.3 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{color}`

This extension provides the ability to request that any object be displayed in a particular colour.

It requires a suitable 'driver' option to provide the necessary `\special` commands to implement the effects. Thus

Using this extension will have no effect on the output unless used with a dvi-driver that explicitly supports it.

Colours are specified as a ⟨shape⟩ modifier which gives the name of the colour requested. It is applied to the whole of the current ⟨object⟩ whether this be text, an XY-pic line, curve or arrow-tip, or a composite object such as a matrix or the complete picture. However some DVI drivers may not be able to support the colour in all of these cases.

`[`⟨colour name⟩`]`	use named colour
`\newxycolor{`⟨name⟩`}{`⟨code⟩`}`	define colour
`\UseCrayolaColors`	load colour names

If the DVI-driver cannot support colour then a request for colour only produces a warning message in the log file. After two such messages subsequent requests are ignored completely.

Named colours and colour models New colour names are created with `\newxycolor`, taking two arguments. Firstly a name for the colour is given, followed by the code which will ultimately be passed to the output device in order to specify the colour. If the current driver cannot support colour, or grayscale shading, then the new name will be recognised, but ignored during typesetting.

For PostScript devices, the XY-ps PostScript dictionary defines operators `rgb`, `cmyk` and `gray` corresponding to the standard RGB and CMYK colour models and grayscale shadings. Colours and shades are described as: r g b `rgb` or c m y k `cmyk` or s `gray`, where the parameters are numbers in the range $0 \le r, g, b, c, m, y, k, s \le 1$. The operators link to the built-in colour models or, in the case of `cmyk` for earlier versions of PostScript, give a simple emulation in terms of the RGB model.

Saving colour and styles When styles are saved using `[=`⟨word⟩`]`, see , then the current colour setting (if any) is saved also. Subsequent use of `[`⟨word⟩`]` recovers the colour and accompanying line-style settings.

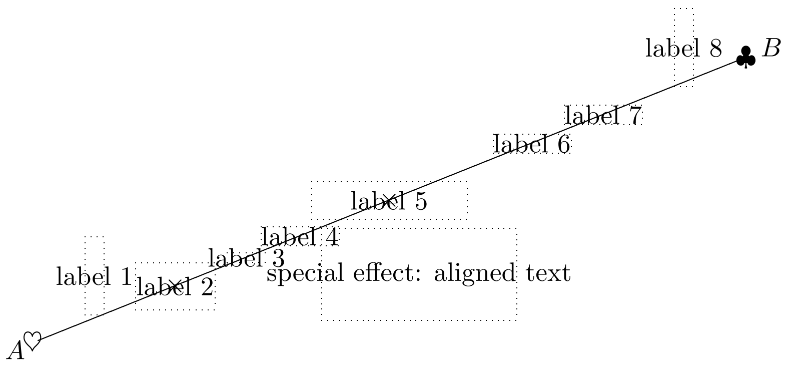


Figure 10: Rotations, scalings and flips

Further colour names are defined by the command `\UseCrayolaColours` that loads the `crayon` option, in which more colours are defined. Consult the file `xyps-col.doc` for the colours and their specifications in the RGB or CMYK models.

xycrayon.tex: This option provides the command to install definitions for the 68 colours recognised by name by Tomas Rokicki's `dvips` driver [11]. This command must be called from a ⟨driver⟩-file which can actually support the colours.

14 Pattern and Tile extension

Vers. 3.4 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{tile}`

This extension provides the ability to request that a filled region be tiled using a particular pattern.

This is an effect not normally available within TEX. Instead it requires a suitable ⟨driver⟩ option to provide the necessary `\special` commands, together with any extra commands needed to implement the effects. Thus

Using this extension will have no effect on the output unless used with a dvi-driver that explicitly supports it.

All effects defined in the `tile` extension can be implemented using most PostScript ⟨driver⟩s, loaded as `\xyoption{`⟨driver⟩`}`.

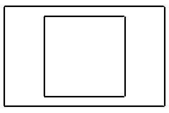
Patterns Patterns are specified as a ⟨shape⟩ modifier, similar to the way colours are specified by name. The pattern is applied to the whole of the current ⟨object⟩ whether this be text, an XY-pic line, curve or arrow-tip, or a composite object such as a matrix or the complete picture. However some DVI-drivers may not support use of patterns in all cases.

If the current DVI-driver cannot support patterns then a request for one simply produces a warning message in the log file. After two such messages subsequent requests are ignored completely.

`[`⟨name⟩`]`	use named pattern
`\newxypattern{`⟨name⟩`}{`⟨data⟩`}`	specify new pattern using ⟨data⟩
`\UsePatternFile{`⟨file⟩`}`	sets default file for patterns
`\LoadAllPatterns{`⟨file⟩`}`	load all patterns in ⟨file⟩
`\LoadPattern{`⟨name⟩`}{`⟨file⟩`}`	load named pattern from ⟨file⟩
`\AliasPattern{`⟨alias⟩`}{`⟨name⟩`}{`⟨file⟩`}`	let ⟨alias⟩ denote pattern from ⟨file⟩.

Although pattern data may be specified directly using `\newxypattern`, it is more usual to load it from a ⟨file⟩ in which many patterns are defined by name, each on a separate line. By convention such files always end in `.xyp` (XY-pattern) so no extension should be specified. The pattern is then requested using either the name supplied in the file or by an alias. Once `\UsePatternFile` has been used, then a null ⟨file⟩ argument to the other commands will still find patterns in the default file. The default remains in effect for the current level of TEX grouping.

For example, the following picture



uses 'filled' frames from the `frame` feature:

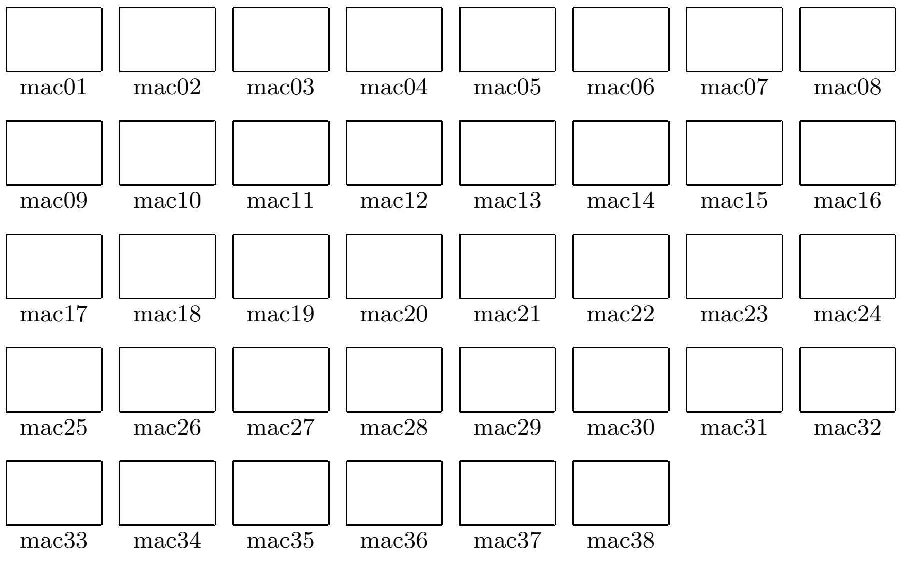


Figure 11: The 38 standard Macintosh patterns.

```
\AliasPattern{bricks}{mac12}{xymacpat}
\AliasPattern{bars}{mac08}{xymacpat}
\xy *+<5pc,3.1pc>{},{*[bricks]\frm{**}}
 ,*+<2.5pc>[o]{},*[bars]\frm{**}
\endxy
```

Pattern data A region is tiled using copies of a single 'cell' regularly placed so as to seamlessly tile the entire region. The ⟨data⟩ appearing as an argument to `\newxypattern` is ultimately passed to the dvi-driver.

The simplest form of pattern data is: ⟨num⟩ ⟨Hex-data⟩, where the data is a 16-character string of Hexadecimal digits; i.e. 0–9, *A*–*F*. Each Hex-digit equates to 4 binary bits, so this data contains 64 bits representing pixels in an 8×8 array. The ⟨num⟩ is an integer counting the number of '0's among the 64 bits. Taken as a fraction of 64, this number or its complement, represents the average density of 'on' pixels within a single cell of the pattern. Drivers unable to provide the fine detail of a pattern may simply use this number, or its complement, as a gray-level or part of a colour specification for the whole region to be tiled.

The file `xymacpat.xyp` contains defining data for the 38 standard patterns available with the Macintosh Operating system. Figure 11 displays all these patterns.

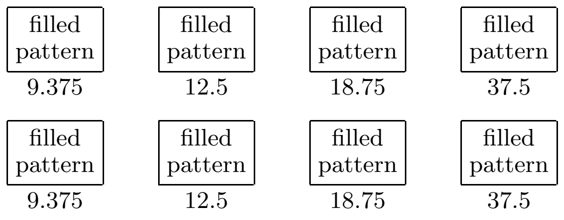
Rotating and Resizing Patterns Some implementations of patterns are sufficiently versatile to allow extra parameters to affect the way the pattern data is interpreted. PostScript is one such implementation in which it is possible to rotate the whole pattern and even to expand or contract the sizes of the basic cell.

Due to the raster nature of output devices, not all such requests can be guaranteed to produce aesthetic results on all devices. In practice only rotations through specific angles (e.g 30°, 45°, 60°) and particular scaling ratios can be reliably used. Thus there is no sophisticated interface provided by XY-pic to access these features. However the 'PostScript escape' mechanism does allow a form of access, when a PostScript ⟨driver⟩ is handling pattern requests.

Special PostScript operators `pa` and `pf` set the pattern angle (normally 0) and 'frequency' measured in *cells per inch*. Hence, when used as an ⟨object⟩-modifier, `[! 30 pa 18.75 pq]` rotates the pattern by 30° clockwise and uses a smaller pattern cell (larger frequency). The default frequency of $12.5 = 300/(8 \times 3)$ means that each pixel in a pattern cell corresponds, on a device of resolution 300dpi, to a 3×3 square of device pixels; on such a device 18.75 uses 2×2 squares.

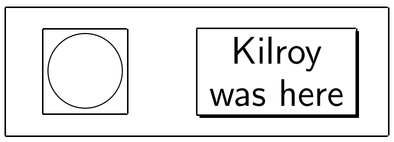
At 300dpi a frequency of $9.375 = 300/(8 \times 4)$ uses 4×4 squares. These match the natural size for pixels on a 75dpi screen and are pretty close for 72dpi screens. Though appropriate for screen displays, these are 'too chunky' for high quality printed work. Doubling the frequency is too fine for some patterns, hence the intermediate choice of 12.5 as default. In order for printed output to match the screen view, a PostScript operator `macfreq` has been defined to facilitate requests for 9.375, via `[!macfreq]`.

The next diagram displays changes to the frequency.



Saving patterns: When styles are saved using ⟨word⟩`]`, see note 4k of §4, then the current pattern (if any) is also saved. Subsequent use of `[`⟨word⟩`]` recovers the pattern as well as colour and line-style settings. This includes any explicit variations applied using the "Style Escape" mechanism.

Here is a variation of an earlier example, with extra effects.



```
\UsePatternFile{xymacpat}
\AliasPattern{bricks}{mac12}{}
\LoadPattern{mac28}{}\LoadPattern{mac05}{}
\xy *=0[! macfreq -45 pa][mac28][|=Bars]{}
,*+<12pc,4pc>{}*[bricks]\frm{**}
,-<3.5pc,0pt>,*+<2.65pc>[o]{},*[Bars]\frm{**}
,*[thicker]\frm{o},+<6pc,0pt>
,*+<5pc, 2.7pc>{},*[mac05]\frm{**},*\frm{-,}
,*[white]\txt\Large\bf\sf{Kilroy\\was here}
\endxy
```

15 Import graphics extension

Vers. 3.6 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{import}`

This feature provides the ability to easy add labels and annotations to graphics prepared outside TEX or LATEX. An XY-pic graphics environment is established whose coordinates match that within the contents of the imported graphic, making it easy to specify exactly where a label should be placed, or arrow drawn to highlight a particular feature.

A command `\xyimport` is defined which is used, in conjunction with imported graphics, to establish a coordinate system appropriate to the particular graphics. This enables ⟨pos⟩itions within the graphic to be easily located, either for labelling or adding extra embellishing features. It is used in either of the follow ways:

`\xyimport(`*width*`,`*height*`){`⟨graphic⟩`}`

`\xyimport(`*width*`,`*height*`)(`*x-off*`,`*y*-off`){`⟨graphic⟩`}`

Normally the ⟨graphics⟩ will be a box containing a graphic imported using the commands from packages such as `graphics`, `epsf` or `epsfig`, or using other commands provided by the local TEX implementation. However the ⟨graphic⟩ could be *any* balanced TEX material whatsoever; provided it occupies non-zero size, both vertically and horizontally.

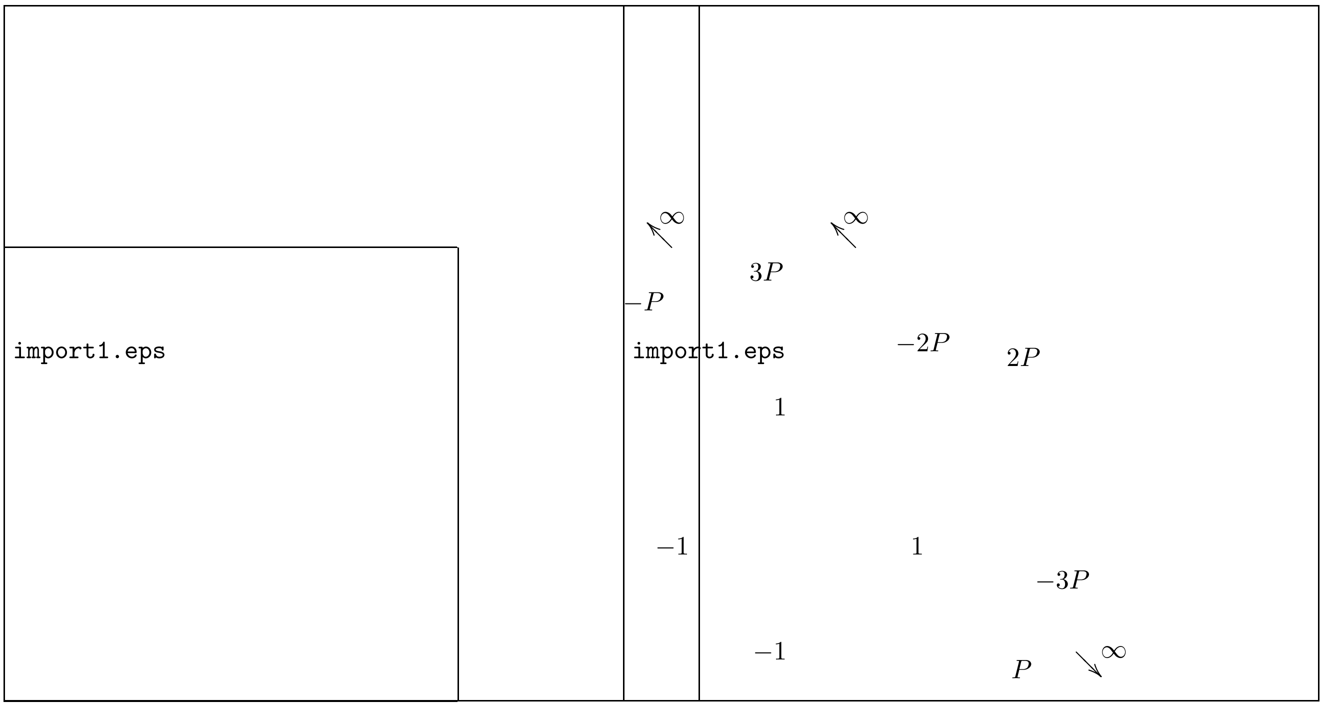
The *width* and *height* are ⟨number⟩s given in the coordinate system for the *contents of the* ⟨graphics⟩. These are not dimensions, but coordinate-lengths, using the units appropriate to the picture displayed by ⟨graphic⟩.

When provided, `(`*x-off*`,`*y*-off`)` give the distance in coordinate units from bottom-left corner to where the origin of coordinates should be located, usually within area covered by the ⟨graphic⟩. Usually the negatives of these numbers will give the coordinate location of the bottom-left corner of the ⟨graphic⟩. If no offsets are supplied then the origin is presumed to lie at the bottom-left corner.

Normally the `\xyimport` command is used at the beginning of an `\xy..\endxy` environment. It is not necessary to give any basis setup, for this is deduced by measuring the dimensions of the ⟨graphic⟩ and using the supplied *width*, *height* and offsets. The ⟨graphic⟩ itself defines named ⟨pos⟩ called `"import"`, located at the origin and having appropriate extents to describe the area covered by the ⟨graphic⟩. This makes it particularly easy to surround the ⟨graphic⟩ with a frame, as on the left side of figure 12, or to draw axes passing through the origin.

Here is the code used to apply the labelling in figure 12:

```
\def\ellipA{\resizebox{6cm}{!}{%
  \includegraphics{import1.eps}}}
\xy
\xyimport(3.7,3.7)(1.4,1.4){\ellipA}*\frm{-}
,!D+<2pc,-1pc>*+!U\txt{%
 framed contents of graphics file}\endxy
\qquad\qquad
\xy\xyimport(3.7,3.7)(1.4,1.4){\ellipA}
,!D+<2pc,-1pc>*+!U\txt{Rational points
 on the elliptic curve: $x^3+y^3=7$}
,(1,0)*+!U{1},(-1,0)*+!U{-1}
,(0,1)*+!R{1},(0,-1)*+!R{-1}
,(2,-1)*+!RU{P},(-1,2)*+!RU{-P}
,(1.3333,1.6667)*+!UR{-2P}
,(1.6667,1.3333)*!DL{\;2P}
,(-.5,1.9)*++!DL{3P},(1.9,-.5)*!DL{\;-3P}
,(-1,2.3)*+++!D{\infty}*=0{},{\ar+(-.2,.2)}
,(.5,2.3)*+++!D{\infty}*=0{},{\ar+(-.2,.2)}
,(2.3,-1)*+++!L{\infty}*=0{},{\ar+(.2,-.2)}
\endxy
```



framed contents of graphics file — Rational points on the elliptic curve: $x^3 + y^3 = 7$

Figure 12: importing a graphic for labelling

This example uses the LaTeX 2$_\varepsilon$ standard `graphics` package to import the graphics file `import1.eps`; other packages could have been used instead. e.g. `epsfig`, `epsf`, or the `\picture` or `\illustration` commands in TEXTURES on the Macintosh.

The only possible problems that can occur are when the graphics package is loaded after XY-pic has been loaded. Generally it is advisable to have XY-pic loading *after* all other macro packages.

16 Movie Storyboard extension

Vers. 3.5 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩
Load as: `\xyoption{movie}`

This extension interprets the `\scene` primitive of the `movie` class, setting the progress indicators to dummy values. The following assumes that your are familiar with the `movie` class.

The size of the frame is determined by the command

`\MovieSetup{width=`*width*`,height=`*height*`,`. . . `}`

(the . . . indicate the other arguments required by the `movie` class but silently ignored by the XY-pic `movie` extension).

Note: This extension still experimental and subject to change. The only documentation is in the `movie.cls` source file.

17 PostScript backend

Vers. 3.7 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{ps}`

XY-ps is a 'back-end' which provides XY-pic with the ability to produce DVI files that use POSTSCRIPT `\specials` for drawing rather than the XY-pic fonts.

In particular this makes it possible to print XY-pic DVI files on systems which do not have the ability to load the special fonts. The penalty is that the generated DVI files will only function with one particular DVI driver program. Hence whenever XY-ps is activated it will warn the user:

XY-pic Warning: The produced DVI file is *not portable*: It contains POSTSCRIPT `\specials` for ⟨one particular⟩ driver

A more complete discussion of the pros and cons of using this backend is included below.

17.1 Choosing the DVI-driver

Including `\xyoption{ps}` within the document preamble, tells XY-pic that the POSTSCRIPT alternative to the fonts should be used, provided the means to do this is also specified. This is done by also specifying a dvi-driver which is capable of recognising and interpreting `\special` commands. Although the file `xyps.tex` is read when the option request is encountered, the macros contained therein will have no effect until an appropriate driver has also been loaded.

With LaTeX 2$_\varepsilon$ both the backend and driver may be specified, along with other options, via a single `\usepackage` command, see [4, page 317]; e.g.

```
\usepackage[ps,textures,color,arrow]{xy}
```

The rebindings necessary to support PostScript are not effected until the `\begin{document}` command is encountered. This means that an alternative driver may be selected, by another `\xyoption{`⟨driver⟩`}`, at any time until the `\begin{document}`. Only the macros relevant to last named ⟨driver⟩ will actually be installed.

The following table describes available support for PostScript drivers. Please consult the individual driver sections in part IV for the exact current list. For each ⟨driver⟩ there is a corresponding file named `xy`⟨driver⟩`.tex` which defines the necessary macros, as well as a documentation file named `xy`⟨driver⟩`.doc`. The spelling is all lower-case, designed to be both descriptive and unique for the 1st 8 characters of the file names.

⟨driver⟩	Description
`dvips`	Tomas Rokicki's DVIPS
`dvips`	Karl Berry's `dvipsk`
`dvips`	Thomas Kiffe's DVIPS for Macintosh
`textures`	Blue Sky Research's TEXTURES v1.7+
`16textures`	Blue Sky Research's TEXTURES v1.6
`oztex`	Andrew Trevorrow's OzTeX v1.8+
`17oztex`	Andrew Trevorrow's OzTeX v1.7

Other DVI-drivers may also work using one of these files, if they use conventions similar to `dvips`, OzTeX or TEXTURES. Alternatively it should not be too difficult to write the files required, using these as a basis indicating the type of information needed to support the specific `\special` commands. Anyone attempting to do this should inform the author and convey a successful implementation to him for inclusion within the XY-pic distribution.

Note: In some previous versions of XY-pic the PostScript backend and driver were loaded simultaneously by a command of the form `\UsePSspecials{`⟨driver⟩`}`. For backward-compatibility these commands should still work, but now loading the latest version of the given ⟨driver⟩. However their future use is discouraged in favour of the option-loading mechanism, via `\xyoption{`⟨driver⟩`}`. This latter mechanism is more flexible, both in handling upgrades of the actual driver support and in being extensible to support more general forms of `\special` commands.

Once activated the PostScript backend can be turned off and on again at will, using the user following commands:

`\NoPSspecials`	cancels PostScript
`\UsePSspecials {}`	restores PostScript

These obey normal TeX scoping rules for environments; hence it is sufficient to specify `\NoPSspecials` within an environment or grouping. Use of PostScript will be restored upon exiting from the environment.

17.2 Why use PostScript

At some sites users have difficulty installing the extra fonts used by XY-pic. The `.tfm` files can always be installed locally but it may be necessary for the `.pk` bitmap fonts (or the `.mf` METAFONT fonts) to be installed globally, by the system administrator, for printing to work correctly. If PostScript is available then XY-ps allows this latter step to be bypassed.

Note: with XY-ps it is still necessary to have the `.tfm` font metric files correctly installed, as these contain information vital for correct typesetting.

Other advantages obtained from using XY-ps are the following:

- Circles and circle segments can be set for arbitrary radii.
- solid lines are straighter and cleaner.
- The range of possible angles of directionals is greatly increased.
- Spline curves are smoother. True dotted and dashed versions are now possible, using equally spaced segments which are themselves curved.
- The PostScript file produced by a driver from an XY-ps DVI file is in general significantly smaller than one produced by processing an 'ordinary' DVI file using the same driver. One reason for this is that no font information for the XY-pic fonts is required in the PostScript file; this furthermore means that the use of XY-pic does not in itself limit the PostScript file to a particular resolution.[9]
- The latest version of XY-pic now enables special effects such as variable line thickness, gray-level and colour. Also, rotation of text and (portions of) diagrams is now supported with some drivers. Similarly whole diagrams can be scaled up or down to fit a given area on the printed page. Future versions will allow the use of regions filled with colour and/or patterns, as well as other attractive effects.

[9]Most TeX PostScript drivers store the images of characters used in the text as bitmaps at a particular resolution. This means that the PostScript file can only be printed without loss of quality (due to bitmap scaling) at exactly this resolution.

Some of the above advantages are significant, but they come at a price. Known disadvantages of using XY-ps include the following:

- A DVI file with specials for a particular POSTSCRIPT driver can only be previewed if a previewer is available that supports exactly the same `\special` format. A separate POSTSCRIPT previewer will usually be required.

  However recent versions of `xdvi` support viewing of POSTSCRIPT using either the GhostScript program or via "Display PostScript". The POSTSCRIPT produced by XY-ps can be viewed this way

- DVI files created using XY-ps in fact lose their "device-independence". So please do not distribute DVI files with POSTSCRIPT specials—send either the TEX source code, expecting the recipient to have XY-pic ☺, or send a (compressed) POSTSCRIPT file.

The latter comment applies to files in which any special 'back-end' support is required, not just to POSTSCRIPT. Of course it can be ignored when you know the configuration available to the intended recipient.

POSTSCRIPT **header file:** With some DVI-drivers it is more efficient to have the POSTSCRIPT commands that XY-ps needs loaded initially from a separate "header" file. To use this facility the following commands are available...

```
\UsePSheader {}
\UsePSheader {<filename>}
\dumpPSdict {<filename>}
\xyPSdefaultdict
```

Normally it is sufficient to invoke `\UsePSheader{}`, which invokes the default name of `xy37dict.pro`, referring to the current version of XY-pic. The optional ⟨filename⟩ allows a different file to be used. Placing `\dumpPSdict{..}` within the document preamble causes the dictionary to be written to the supplied ⟨filename⟩.

See the documentation for the specific driver to establish where the dictionary file should be located on any particular TEX system. Usually it is sufficient to have a copy in the current working directory. Invoking the command `\dumpPSdict{}` will place a copy of the requisite file, having the default name, in the current directory. This file will be used as the dictionary for the current processing, provided it is on the correct directory path, so that the driver can locate it when needed. Consult your local system administrator if you experience difficulties.

18 TPIC backend

Vers. 3.3 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{tpic}`

This option allows the XY-pic fonts to be replaced by TPIC `\specials`, when used with a dvi-driver capable of supporting them. Extra capabilities include smoother lines, evenly spaced dotted/dashed curves, variable line-widths, gray-scale fills of circles, ellipses and polygonal regions.

Use of TPIC `\specials` offers an alternative to the XY-pic fonts. However they require a dvi-driver that is capable of recognizing and interpreting them. One such viewer is `xdvik`, Karl Berry's modification to the `xdvi` viewer on UNIX[10] systems running X-windows or a derivative. `dvipsk`, Karl Berry's modification to `dvips` also handles TPIC `\specials`, so `xdvik/dvipsk` is an good combination for quality screen-display and POSTSCRIPT printing.

Once loaded using `\xyoption{tpic}`, with an appropriate ⟨driver⟩ also specified either already or subsequently, the following commands are available to turn the TPIC backend off/on.

`\NoTPICspecials`	turns off TPIC specials.
`\UseTPICspecials`	reinstates TPIC specials.

There is a limit to the number of points allowable in a path. For paths constructed by XY-pic, which includes spline curves, when the limit is reached the path is automatically flushed and a new path commenced. The following command can be used to customise this limit—initially set at 300 for use with XDVI—to suit alternative ⟨driver⟩s.

`\maxTPICpoints{`⟨num⟩`}`	set maximum for paths

Of the curves defined in the `xycurve` extension, only solid spline curves are supported. This is done by treating the spline as a polygon (poly-line) with many segments. The dotted or dashed variants do not work correctly.

Implementations of TPIC draw dashed polygons such that the start and finish of each segment is solid. Since these segments can be very short, the effect is simply to create a solid line. Similarly the shortness of the segments tends to give nothing at all for large portions of a dotted curve. What is needed is an implementation whereby the on/off nature of a dashed or dotted polygon is determined by the accumulated length, not the length along just the current segment.

[10] UNIX is a trademark of Bell Labs.

19 em-TeX backend

Vers. 3.3 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{emtex}`

Eberhard Matte's em-TEX implementation provides a suite of `\special` commands to facilitate the drawing of lines, both on-screen and with various printing devices. This 'back-end' extension allows the lines in XY-pic diagrams to be drawn using these methods.

Note that this extension does not have to be used with em-TEX. Better results may be obtained using the PostScript back-end and DVIPS ⟨driver⟩, since a version of DVIPS is available for em-TEX. However, in particular for screen previewing purposes, it may be convenient to use this back-end. Furthermore note that DVIPS is capable of supporting em-TEX`\special`s.

Once loaded using `\xyoption{emtex}`, with an appropriate ⟨driver⟩ also specified either already or subsequently, the following commands are available to turn the em-TEX backend off/on.

`\NoEMspecials`	turns off em-TEX specials.
`\UseEMspecials`	reinstates em-TEX specials.

Of the curves defined in the `xycurve` extension, only solid spline curves are supported. This is done by treating the spline as a polygon (poly-line) with many segments.

20 Necula's extensions

Vers. 0.0 by George C. Necula ⟨necula@cs.cmu.edu⟩
Load as: `\xyoption{necula}`

This option contains two extensions of the XY-pic kernel: A way to expand TEX macros in object ⟨modifier⟩s, and a way to specify arbitrary polygons as the ⟨shape⟩ of an object.

20.1 Expansion

The special syntax `e|`⟨macros⟩`|` is introduced in an object ⟨modifier⟩s and ⟨coord⟩inates. It expands the given TEX ⟨macros⟩ (with `\edef`) before reinterpretation as a ⟨modifier⟩ of ⟨coord⟩, respectively.

This code may become part of the XY-pic kernel at a certain point.

20.2 Polygon shapes

A polygon ⟨shape⟩ is specified as

`[P:`⟨pos⟩`,`...`,`⟨pos⟩`]`

where `[P:`p_1`,`...`,`p_n`]` denotes the shape obtained by tracking the edge with each p_i a position relative to the object reference point. ⟨vector⟩s and ⟨corner⟩s can be used directly; otherwise use `-p` to get the relative position.

Note: Do not use `{}` or `[]` in the ⟨pos⟩itions.

Bug: The algorithm assumes that the reference point is always inside the polygon.

It is possible to frame polygons is also possible.

Bug: This code should be merged with the 'frame' and 'poly' options.

The example at the end of §**??** illustrates the extensions.

Part III
Features

This part documents the notation added by each standard feature option. For each is indicated the described version number, the author, and how it is loaded.

The first two, 'all' and 'dummy', described in §§21 and 22, are trivial features that nevertheless prove useful sometimes. The next two, 'arrow' and '2cell', described in §23 and 24, provide special commands for objects that 'point'. The following, 'matrix' in §25, 'graph' in §26, 'poly' in §27, and 'knot' in §30, are *input modes* that support different overall structuring of (parts of) XY-pictures.

21 All features

Vers. 3.3 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩
Load as: `\xyoption{all}`

As a special convenience, this feature loads a subset of XY-pic,[11] namely the extensions: `curve` (*cf.* §8), `frame` (§9), `cmtip` (§10), `line` (§11), `rotate` (§12), `color` (§13), and the following features: `matrix` (§25), `arrow` (§23), and `graph` (§26).

22 Dummy option

Vers. 3.3 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩
Load as: `\xyoption{dummy}`

This option is provided as a template for new options, it provides neither features nor extensions but it does count how many times it is requested.

[11] The name 'all' hints at the fact that these were all the available options at the time 'all' was added.

23 Arrow and Path feature

Vers. 3.5 by Kristoffer H. Rose ⟨krisrose@brics.dk⟩
Load as: `\xyoption{arrow}`

This feature provides XY-pic with the arrow paradigm presented in [12].

Note: `\PATH` command incompatibly changed for version 3.3 (the `\ar` command is unaffected).

The basic concept introduced is the *path*: a connection that *starts* from c (the current object), *ends* at a specified object, and may be split into several *segments* between intermediate specified objects that can be individually labelled, change style, have breaks, etc.

§23.1 is about the `\PATH` primitive, including the syntax of paths, and §23.2 is about the `\ar` customisation of paths to draw arrows using XY-pic directional objects.

23.1 Paths

The fundamental commands of this feature are `\PATH` and `\afterPATH` that will parse the ⟨path⟩ according to the grammar in figure 13 with notes below.

Notes

23a. An ⟨action⟩ can be either of the characters `=/`. The associated ⟨stuff⟩ is saved and used to call

`\PATHaction`⟨action⟩`{`⟨stuff⟩`}`

before and *after* each segment (including all ⟨labels⟩) for `=` and `/`, respectively.

The default `\PATHaction` macro just expands to "`\POS` ⟨stuff⟩ `\relax`" thus ⟨stuff⟩ should be of the form ⟨pos⟩ ⟨decor⟩. The user can redefine this—in fact the `\ar` command described in §23.2 below is little more than a special `\PATHaction` command and a clever defaulting mechanism.

23b. It is possible to include a number of default ⟨labels⟩ *before* the ⟨labels⟩ of the actual ⟨segment⟩ are interpreted, using `~`⟨which⟩`{`⟨labels⟩`}`. The specified ⟨which⟩ determines for which segments the indicated ⟨labels⟩ should be prefixed as follows:

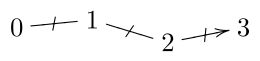
⟨which⟩	applied to...
`<`	next segment only
`>`	last segment only
`=`	every segment

(when several apply to the same segment they are inserted in the sequence `<>+`).

This is useful to draw connections with a 'center marker' in particular with arrows, *e.g.*, the 'mapsto' example explained below can be changed into a 'breakto' example: typing

```
\xy*+{0}\PATH
 ~={**\dir{-}}
 ~>{|>*\dir{>}}
 ~+{|*\dir{/}}
 '(10,1)*+{1} '(20,-2)*+{2} (30,0)*+{3}
\endxy
```

will typeset



Note, however, that what goes into `~+{...}` is ⟨labels⟩ and thus not a ⟨pos⟩ – it is not an action in the sense explained above.

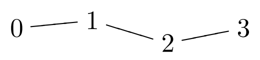
23c. Specifying `~{`⟨stuff⟩`}` will set the "failure continuation" to ⟨stuff⟩. This will be inserted when the last ⟨segment⟩ is expected—it can even replace it or add more ⟨segment⟩s, *i.e.*,

```
\xy *+{0} \PATH ~={**\dir{-}}
  ~{'(20,-2)*+{2} (30,0)*+{3}} '(10,1)*+{1}
\endxy
```

is equivalent to

```
\xy *+{0} \PATH ~={**\dir{-}}
  '(10,1)*+{1} '(20,-2)*+{2} (30,0)*+{3}
\endxy
```

typesetting



because when `\endxy` is seen then the parser knows that the next symbol is neither of the characters `~'`‘ and hence that the last ⟨segment⟩ is to be expected. Instead, however, the failure continuation is inserted and parsed, and the ⟨path⟩ is finished by the inserted material.

Failure continuations can be nested:

```
\xy *+{0} \PATH ~={**\dir{-}}
 ~{~{(30,0)*+{3}}
 '(20,-2)*+{2}} '(10,1)*+{1}
\endxy
```

will also typeset the connected digits.

23d. A "straight segment" is interpreted as follows:

1. First p is set to the end object of the previous segment (for the first segment this is c just before the path command) and c is set to the ⟨pos⟩ starting the ⟨segment⟩, and the current ⟨slide⟩ is applied.

Syntax			Action
`\PATH` ⟨path⟩			interpret ⟨path⟩
`\afterPATH{`⟨decor⟩`}` ⟨path⟩			interpret ⟨path⟩ and then run ⟨decor⟩
⟨path⟩	⟶	`~` ⟨action⟩ `{` ⟨stuff⟩ `}` ⟨path⟩	set ⟨action⟩[23a] to ⟨stuff⟩
	|	`~` ⟨which⟩ `{` ⟨labels⟩ `}` ⟨path⟩	add ⟨labels⟩ prefix for some segments[23b]
	|	`~` `{` ⟨stuff⟩ `}` ⟨path⟩	set failure continuation[23c] to ⟨stuff⟩
	|	`'` ⟨segment⟩ ⟨path⟩	make straight segment[23d]
	|	`` ` `` ⟨turn⟩ ⟨segment⟩ ⟨path⟩	make turning segment[23f]
	|	⟨segment⟩	make last segment[23g]
⟨turn⟩	⟶	⟨diag⟩ ⟨turnradius⟩	1/4 turn[23f] starting in ⟨diag⟩
	|	⟨cir⟩ ⟨turnradius⟩	explicit turn[23f]
⟨turnradius⟩	⟶	⟨empty⟩	use default turn radius
	|	`/` ⟨dimen⟩	set *turnradius* to ⟨dimen⟩
⟨segment⟩	⟶	⟨path-pos⟩ ⟨slide⟩ ⟨labels⟩	segment[23e] with ⟨slide⟩ and ⟨labels⟩
⟨slide⟩	⟶	⟨empty⟩ | `<` ⟨dimen⟩ `>`	optional slide[23h]: ⟨dimen⟩ in the "above" direction
⟨labels⟩	⟶	`^` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	label with ⟨it⟩[23i] *above* ⟨anchor⟩
	|	`_` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	label with ⟨it⟩[23i] *below* ⟨anchor⟩
	|	`|` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	break with ⟨it⟩[23j] at ⟨anchor⟩
	|	⟨empty⟩	no more labels
⟨anchor⟩	⟶	`-` ⟨anchor⟩ | ⟨place⟩	label/break placed relative to the ⟨place⟩ where `-` is a synonym for `<>(.5)`
⟨it⟩	⟶	⟨digit⟩ | ⟨letter⟩ | `{`⟨text⟩`}` | ⟨cs⟩	⟨it⟩ is a default label[23k]
	|	`*` ⟨object⟩	⟨it⟩ is an ⟨object⟩
	|	`@` ⟨dir⟩	⟨it⟩ is a ⟨dir⟩ectional
	|	`[` ⟨shape⟩ `]` ⟨it⟩	use `[`⟨shape⟩`]` for ⟨it⟩
⟨alias⟩	⟶	⟨empty⟩ | `="`⟨id⟩`"`	optional name for label object[23l]

Figure 13: ⟨path⟩s

2. Then the `=` and `<` *segment actions* are expanded (in that sequence) and the `<` action is cleared. The resulting p and c become the *start* and *end* object of the segment.

3. Then all ⟨labels⟩ (starting with the ones defined as described in note 23b below).

23e. A *segment* is a part of a ⟨path⟩ between a previous and a new *target* given as a ⟨path-pos⟩: normally this is just a ⟨pos⟩ as described in §3 but it can be changed to something else by changing the control sequence `\PATHafterPOS` to be something other than `\afterPOS`.

23f. A *turning* segment is one that does not go all the way to the given ⟨pos⟩ but only as far as required to make a turn towards it. The c is set to the actual turn object after a turning segment such that subsequent turning or other segments will start from there, in particular the last segment (which is always straight) can be used to finish a winding line.

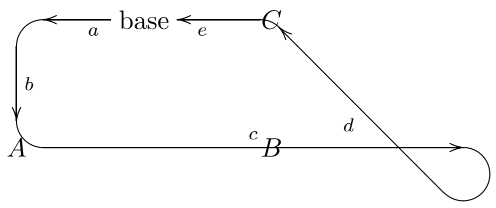
What the turn looks like is determined by the ⟨turn⟩ form:

⟨empty⟩ Nothing between the `` ` `` and the ⟨pos⟩ is interpreted the same as giving just the ⟨diag⟩ last used *out* of a turn.

⟨diag⟩ Specifying a single ⟨diag⟩ d is the same as specifying either of the ⟨cir⟩cles d`^` or d`_`, depending on whether the specified ⟨pos⟩ has its center 'above' or 'below' the line from p in the ⟨diag⟩onal direction.

⟨cir⟩ When a full explicit ⟨cir⟩cle is available then the corresponding ⟨cir⟩cle object is placed such that its ingoing direction is a continuation of a straight connection from p and the outgoing direction points such that a following straight (or last) segment will connect it to c (with the same slide).

Here is an example using all forms of ⟨turn⟩s:



was typeset by

```
\xy <4pc,0pc>:(0,0)
 *+\txt{base}="base"
 \PATH ~={**\dir{-}?>*\dir{>}}
       `l   (-1,-1)*{A} ^a
       `      (1,-1)*{B} ^b
       `_ul  (1, 0)*{C} ^c
       `ul^l "base"      ^d
             "base"      ^e
\endxy
```

Bug: Turns are only really resonable for paths that use straight lines like the one above.

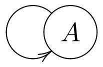
Note: Always write a valid ⟨pos⟩ after a ⟨turn⟩, otherwise any following ^ or _ labels can confuse the parser. So if you intend the ^r in `^r to be a label then write `,^r, using a dummy , ⟨pos⟩ition.

The default used for *turnradius* can be set by the operation

`\turnradius` ⟨add op⟩ `{`⟨dimen⟩`}`

that works like the kernel `\objectmargin` etc. commands; it defaults to 10pt.

Exercise 23: Typeset



using ⟨turn⟩s.

23g. The last segment is exactly as a straight one except that the > action (if any) is executed (and cleared) just after the < action.

23h. "Sliding" a segment means moving each of the p, c objects in the direction perpendicular to the current direction at each.

23i. Labelling means that ⟨it⟩ is dropped relative to the current segment using a ? ⟨pos⟩ition. This thus depends on the user setting up a connection with a ** ⟨pos⟩ as one of the actions—typically the = action is used for this (see note 23d for the details). The only difference between ^ and _ is that they shift the label in the ^ respectively _ direction; for straight segments it is placed in the "superscript" or "subscript" position.

Labels will be separated from the connection by the *labelmargin* that you can set with the operation

`\labelmargin` ⟨add op⟩ `{`⟨dimen⟩`}`

that works like the kernel `\objectmargin` command; in fact *labelmargin* defaults to use *objectmargin* if not set.

23j. Breaking means to "slice a hole" in the connection and insert ⟨it⟩ there. This is realized by typesetting the connection in question in *subsegments*, one leading to the break and one continuing after the break as described in notes 23a and 23d.

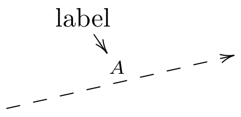
The special control sequence `\hole` is provided to make it easy to make an empty break.

23k. Unless ⟨it⟩ is a full-fledged ⟨object⟩ (by using the * form), it is typeset using a `\labelbox` object (initially similar to `\objectbox` of basic Xy-pic but using `\labelstyle` for the style).

Remark: You can only omit the {}s around single letters, digits, and control sequences.

23l. A label is an object like any other in the Xy-picture. Inserting an ⟨alias⟩ ="⟨id⟩" saves the label object as "⟨id⟩" for later reference.

Exercise 24: Typeset



23.2 Arrows

Arrows are paths with a particularly easy syntax for setting up arrows with *tail*, *stem*, and *head* in the style of [12]. This is provided by a single ⟨decor⟩ation the syntax of which is described in figure 14 (with the added convention that a raised '*' means 0 or more repetitions of the preceeding nonterminal).