XY-pic Reference Manual

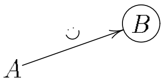
Kristoffer H. Rose ⟨kris@diku.dk⟩[×]
Ross Moore ⟨ross@mpce.mq.edu.au⟩[†]

Version 2.12/3β[‡] ⟨1994/10/25⟩

Abstract

This manual summarises the capabilities of the XY-pic package for typesetting graphs and diagrams in TEX.

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

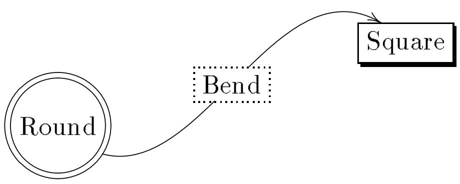


was drawn by the XY-pic kernel code

```
\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" **i\crvs{},
   ?*+\hbox{Bend}="b"*\frm{.} ?>*\dir{>},
 "o";"s"."b" **\crvs{-},
 "o"."b";"s" **\crvs{-}
\endxy
```

using the 'curve' and 'frame' extensions.

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

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

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

```
\xygraph{~{(1.5,0):(0,.7)::}
 []Z        ( [u]X :_f ? , [l]Y :^g ? )
 [ul]{X \times_Z Y}="xy"
            ( ? :_p "X" , ? :^q "Y" )
 [ul]U      ( ? :@/^.5pc/ ^x "X" ,
              ? :@/_.5pc/ _y "Y" ,
              ? :@{-->}   |{(x,y)} "xy" ) }
```

We will not describe the combination of features in this manual: refer to the User's Guide [14] for a tutorial on how diagrams like the above can be typeset.

The current implementation is programmed completely within "standard TEX and METAFONT", *i.e.*, using TEX macros (no `\specials`) and fonts designed using METAFONT. Optionally a special 'backend' makes it possible to produce DVI files with 'specials' for PostScript[1] drivers.

[×] DIKU (Computer Science dept.), University of Copenhagen, Universitetsparken 1, DK–2100 København Ø, Denmark.

[†] MPCE (Mathematics dept.), Macquarie University, North Ryde, Sydney, Australia NSW 2109.

[‡] The "/3β" in the version is meant to indicate that this is a β-test version of XY-pic version 3 currently under development by the authors, thus this manual contains a few 'Bug' and 'To Do' paragraphs describing facilities not yet fully implemented.

Partial funding for this project has been provided by a Macquarie University Research Grant (MURG), by the Australian Research Council (ARC), and through a research agreement with the Digital Equipment Corporation (DEC).

[1] PostScript is a trademark of Adobe, Inc.

Contents

& &&& &&&&&&

&& E&&&&&&&&&&

&&& F&&&&&&&

List of Figures

Part I

Bhe Bernel

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.dk⟩

After giving an overview of the Xy-pic environment in §1 we 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 in this part but in the complete TEXnical documentation [11].

&&&&&&&&

We will 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.

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 in order to experiment with it.

1.1 L&&&&&&p&c

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 [8]).

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 [11] 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 `\dir{-}`) are only checked to be different from `\relax`.

Category codes: Unfortunately the situation is complicated by the flexibility of TEX's input format. The culprit is the 'category code' concept of TEX (*cf.* [5, 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).

Integration with standard formats The integration with various formats is handled by the `xyidioms.tex` file and the integration as a LATEX [8] package by `xy.sty`:

xyidioms.doc: This included file provides 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 (2.12) handles plain TEX (version 2 and 3 [5]), AMS-TEX (version 2.0 and 2.1 [15]), LATEX (version 2.09 [7] and 2ε [8]), AMS-LATEX (version 1.0, 1.1 [1], and 1.2), and eplain (version 2.6 [2])[3].

xy.sty: If you use LATEX then this file makes it possible to load Xy-pic as a 'package' using the LATEX 2ε [8]

---

[2] BNF is the notation for "meta-linguistic formulae" first used in [9] to describe the syntax of the Algol programming language. We use it with the conventions of the TEXbook [5]: '⟶' 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 '←' should be read 'is copied from'.

[3] Although there is a name conflict between the 'v2' feature and eplain that both define `\arrow`.

`\usepackage` command:

`\usepackage [`⟨option⟩`,...] {xy}`

---

where the ⟨option⟩s will be interpreted as if passed to `\xyoption` (*cf.* §7); furthermore options that require special activation will also be activated when loaded this way (*e.g.*, including `cmtip` in the ⟨option⟩ list will not only perform `\xyoption {cmtip}` but also `\UseComputerModernTips`).

Driver package options (*cf.* [3, table 11.2, p.317]) will invoke the appropriate backend (*cf.* §22).

The file also works as a LaTeX 2.09 [7] 'style option' although you will have to load options with the XY-pic mechanism.

1.2 L&&&, &&&&&&&, &&& &&&&&&&&

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 F&&&&

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 A&&&c&&&&&&

One final thing that you must be aware of is the fact 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 XY-pic v.2.12 kernel allocates 6 counters, 27 dimensions, 2 box registers, 3 token registers, 1 read channel, and 1 write channel (when running under plain TEX; under LaTeX and AMS-TEX slightly less is allocated because the provided temporaries are used). Options may allocate further registers.

2 Picture basics

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

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

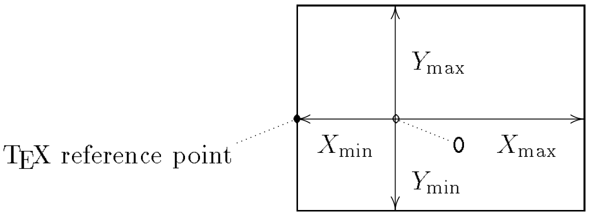
`\xy` ⟨pos⟩ ⟨decor⟩ `\endxy`

---

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 &&&&&&&&&

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:



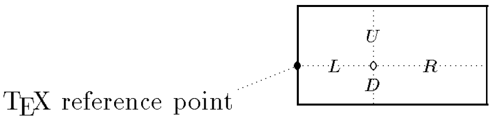
---

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

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.

2.2 &&&&c&&

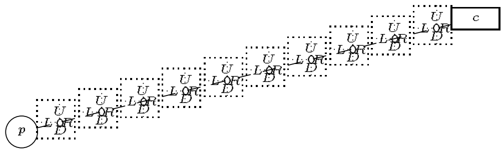
The simplest form of putting things into the picture is to 'drop' an *object* at a position. An object is like a TEX box except that it has a general *Edge* around its reference point—in particular this has the *extents* (*i.e.*, it is always contained within) the dimensions L, R, U, and D away from 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 C&&&&c&&&&&

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 ᴛ&c&&&&&&&

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 &&&&p&c &&&&&

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}`>`, and accessed by the usual notation using parenthesis:

$$\begin{aligned}(x,y) \;=\; < \; & X_{origin} + x \times X_{xbase} + y \times X_{ybase} \;\;, \\ & Y_{origin} + x \times Y_{xbase} + y \times Y_{ybase} \;\;\; >\end{aligned}$$

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 dimension parameters are directly available as TEX `\dimen` registers with the obvious names: `\Xmin`, `\Xmax`, `\Ymin`, and `\Ymax`; `\Xp`, `\Yp` `\Dp`, `\Up`, `\Lp`, and `\Rp`; `\Xc`, `\Yc` `\Dc`, `\Uc`, `\Lc`, and `\Rc`; `\Xorigin`, `\Yorigin`, `\Xxbase`, `\Yxbase`, `\Xybase`, and `\Yybase`.

The edges are not directly available (but see the technical documentation for how to access them).

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 object inherits the size of the ⟨coord⟩, *i.e.*,

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

Figure 1: ⟨pos⟩itions.

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.

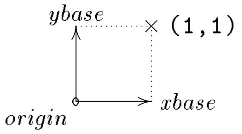
Exercise 3: What does the ⟨pos⟩ ... `!R-L` do?

Bug: The result of `!` is always a rectangle currently.

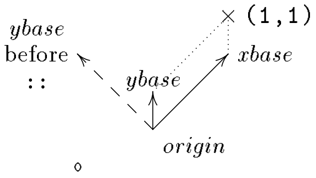
3c. A ⟨pos⟩ *covers* another if it is a rectangle with size sufficiently large that the other is "underneath". The `.` operation "extends" a ⟨pos⟩ to cover an additional one—the reference point of c is not moved but the shape is changed to a rectangle such that the entire p object is covered.

Note: 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 on 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 `0;<1cm,0cm>:` will set the coordinate system



and `<1cm,.5cm>; <2cm,1.5cm>: <1cm,1cm>::` will define



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

When working with vectors these two special ⟨factor⟩s are particularly useful:

`\halfroottwo`	$0.70710678 \approx \sqrt{2}/2$
`\halfrootthree`	$0.86602540 \approx \sqrt{3}/2$

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 to do something that typesets something a `<`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 is something that typesets something connecting p and c sets the `?` ⟨pos⟩ operator up accordingly.

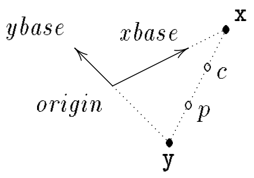
3h. Using `?` will "pick a place" along the most recent connection typeset with `**`. What exactly this means is determined by the object that was used for the connection and by the modifiers described in general terms here.

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).

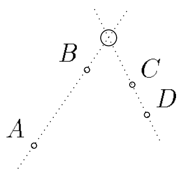
Finally, the ⟨slide⟩ will move the position a dimension further along the connection at the picked position. For straight connections (the only ones kernel XY-pic provides) this is the same as adding a vector in the tangent direction, *i.e.*, `?`.../A/ is the same as `?`...`+/`A`/`.

All this is probably best illustrated with some examples: each ⊗ in figure 2 is typeset by a sequence of the form p`;` c `**\dir{.} ?`⟨place⟩ `*{\oplus}` where we indicate the `?`⟨place⟩ in each case.

3i. 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. These are probably best illustrated by the following example where they are shown for a coordinate system and a p, c pair:



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 as the point marked with a large circle here:



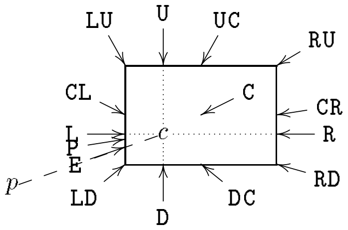
3j. A ⟨pos⟩ ⟨decor⟩ *grouped* in `{}`-braces is interpreted in a local scope in the sense that any p and *base* built within it are forgotten afterwards. **Remark:** Only p and *base* are restored—it is not a TEX group.

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

3k. 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 (`**`) or subsequent placement (`?`) position.

It is possible to give a ⟨direction⟩ as described in the next section (figure 3 and note 4k in particular) that will then be used to set the value of α.

3l. 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 'proportional' point `P` is computed in a complex way to make the object look 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

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

typeset? *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.

3m. The *stack* is a special construction useful for storing a sequence of ⟨pos⟩itions. `@i` initialises, *i.e.*, clears the stack such that it contains no positions, `@+` 'pushes' c onto it, *i.e.*, adds on the 'top' of the stack, increasing the 'depth' by one, and `@-` 'pops' the top element off the stack, decreasing the depth by one. It is an error to pop when the stack is empty.

The special ⟨coord⟩inates `s`n, where n is either a single digit or a positive integer in `{}`s, refer to the n'th position *below the top*, *i.e.*, `s0` is the position on the top, `s1` the one below that, etc.

Exercise 8: Assume the positions A, B, C, and D are defined. What does the stack contain after the ⟨pos⟩ition `@i,` A`@+,` B`@+, @-,` C`,` D`@+` ?

Furthermore, `@(` 'hides' the current stack and creates a fresh stack that can be used as above and once it has served its purpose `@)` will purge it and reestablish the saved stack (issuing a warning message if the purged stack is non-empty).

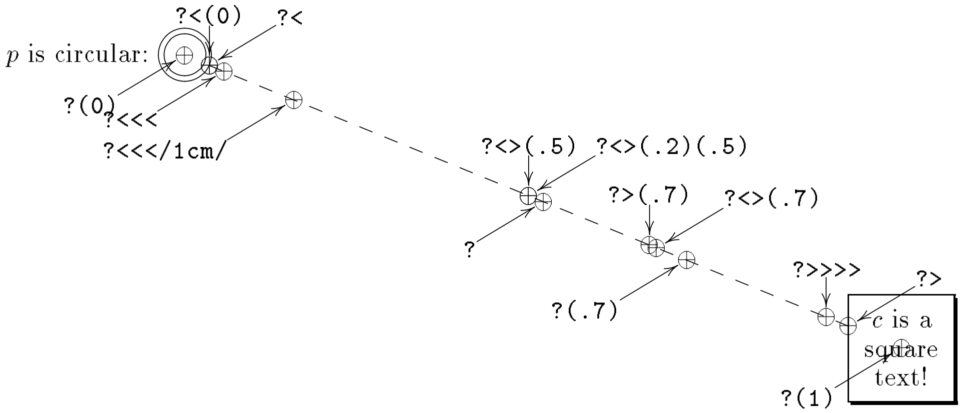


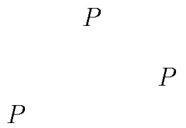
Figure 2: Example ⟨place⟩s

3n. To 'do ⟨coord⟩ for every stack element' means to set *c* to all the elements of the stack, 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.

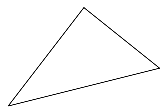
This can be used to repeat a particular ⟨coord⟩ for several points:

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

will typeset



Exercise 9: How would you change the above to connect the points as shown below?



3o. 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, *i.e.*, "⟨id⟩" always refers to the last thing saved with that ⟨id⟩.

Note: There is no distinction between ⟨id⟩s used for saved coordinates and for macros and described in the next note.

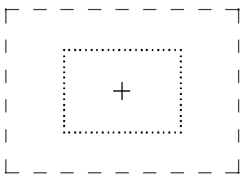
3p. The general form, =⟨code⟩"⟨id⟩" can be used to save various things:

⟨code⟩	effect
:	"⟨id⟩" restores current *base*
⟨coord⟩	"⟨id⟩" interprets ⟨coord⟩

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 10: 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:



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.

To Do: Explain how strange TEX error messages (first of all `box expected`) can result from incomplete ⟨object⟩ specifications.

Notes

4a. A default ⟨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` it should be in math mode.

4b. An ⟨object⟩ built from a TEX box with dimensions $w \times (h + d)$ will have $L_c = R_c = w/2$, $H_c = D_c = (h + d)/2$, 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 11: 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 posible ⟨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
`+`	`+<`$2 \times \mathit{objectmargin}$`>`
`-`	`-<`$2 \times \mathit{objectmargin}$`>`
`=`	`=<`$\mathit{objectwidth}$`,`$\mathit{objectheight}$`>`
`+=`	`+=<`$\max(L_c + R_c, D_c + U_c)$`>`
`-=`	`-=<`$\min(L_c + R_c, D_c + U_c)$`>`

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 12: 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.

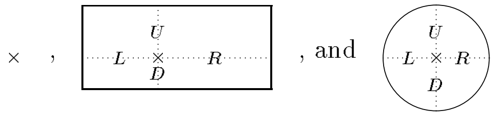
4h. An *invisible* object will be treated completely normal except that it won't be typeset, *i.e.*, XY-pic will behave as if it was.

4i. 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.

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⟩	use ⟨library object⟩ (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] with the right size
⟨modifier⟩	⟶	`!` ⟨vector⟩	⟨object⟩ has its is reference point shifted[4f] by ⟨vector⟩
	|	`!`	⟨object⟩ has the original reference point reinstated
	|	⟨add op⟩ ⟨size⟩	change ⟨object⟩ size[4g]
	|	`i` | `h`	⟨object⟩ is invisible[4h], hidden[4i]
	|	`[` ⟨shape⟩ `]`	⟨object⟩ is given the specified ⟨shape⟩[4j]
	|	⟨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 surrounding the ⟨vector⟩
⟨direction⟩	⟶	⟨diag⟩	⟨diag⟩onal direction[4k]
	|	`v` ⟨vector⟩	direction[4k] of ⟨vector⟩
	|	⟨direction⟩ `:` ⟨vector⟩	vector relative to ⟨direction⟩[4k]
	|	⟨direction⟩ `_` | ⟨direction⟩ `^`	90° clockwise/anticlockwise of ⟨direction⟩[4k]
⟨diag⟩	⟶	⟨empty⟩	default diagonal[4k]
	|	`l` | `r` | `d` | `u`	left, right, down, up diagonal[4k]
	|	`ld` | `rd` | `lu` | `ru`	left/down, … diagonal[4k]
⟨composite⟩	⟶	⟨object⟩	first object is required
	|	⟨composite⟩ `*` ⟨object⟩	add ⟨object⟩ to composite object box[4d]

Figure 3: ⟨object⟩s.

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



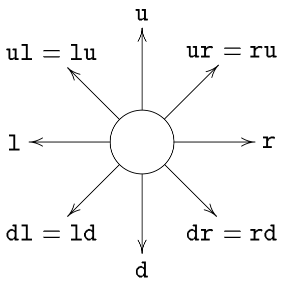
where the × denotes the point of the reference position in the object (the first is a point). Extensions can provide more shapes, however, all shapes set the extent dimensions L, R, D, and U.

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

Note: Extensions may add ⟨shape⟩ object ⟨modifier⟩s of two kinds: either `[`⟨keyword⟩`]` or `[`⟨character⟩ ⟨argument⟩`]`. Some of these ⟨shape⟩s do other things than set the edge of the object.

4k. Setting the current direction is simply pretending for the typesetting of the object (and the following ⟨modifier⟩s) that some connection set it.

It is particularly easy to set absolute, ⟨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`)`. With the initial coordinate system this means that the directions `ur` and `v(1,1)` are identical.

The action for a `v` reads a ⟨vector⟩ and sets the direction accordingly using some expansion hackery to propagate it out. The *origin* is cleared locally to make `v(`x`,`y`)` behave as it should, *i.e.*, use the direction of

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

Adding `_` and `^` denote the result of rotating the default direction a right angle in the positive and negative direction.

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

To Do: Allow `:a(`⟨angle⟩`)`.

Exercise 13: What is the effect of the ⟨modifier⟩s `v/1pc/` 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 used *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**{}?>*\dir2{>}**\dir2{-}
 \restore};p,}
```

will make the code `\ToPOS` ⟨pos⟩ make a double 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 nontrivial ways. Check the TEX-nical documentation [11] for details about this!

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

Syntax			Action
⟨decor⟩	⟶	⟨command⟩ ⟨decor⟩	either there is a command…
	|	⟨empty⟩	…or there isn't.
⟨command⟩	⟶	`\save` ⟨pos⟩	save state^{5a} for restoration by later `\restore`, 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 spaces!) nor changes the grouping may be used
	|	`\xyverbose` | `\xytracing` | `\xyquiet`	tracing^{5c} commands
	|	`\xyignore {`⟨pos⟩ ⟨decor⟩`}`	ignore^{5d} Xy-code
	|	`\xycompileto {`⟨name⟩`} {`⟨pos⟩ ⟨decor⟩`}`	compile^{5e} to file ⟨name⟩`.xyc`

Figure 4: ⟨decor⟩ations.

5c. `\xyverbose` will switch on a tracing of all the Xy-pic commands executed. `\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 the commands to generate parts of an Xy-picture to a file such that subsequent typesetting of those parts is significantly faster: this is called *compiling*. The created file will be named ⟨name⟩`.xyc` and contain code to check that the compiled code still corresponds to the ⟨pos⟩ ⟨decor⟩ as well as more efficient compiled code to redo it. If the ⟨pos⟩ ⟨decor⟩ has changed then the compilation is redone and ⟨name⟩`.xyc` recreated.

Bug: Currently you can only compile matrices (built with the matrix feature) where all entries are empty or start with something that is unexpandable.

6 Kernel object library

In this section we present the *library objects* provided with the kernel language—several options add 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 ┳&&&c&&&&&&&

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⟩).

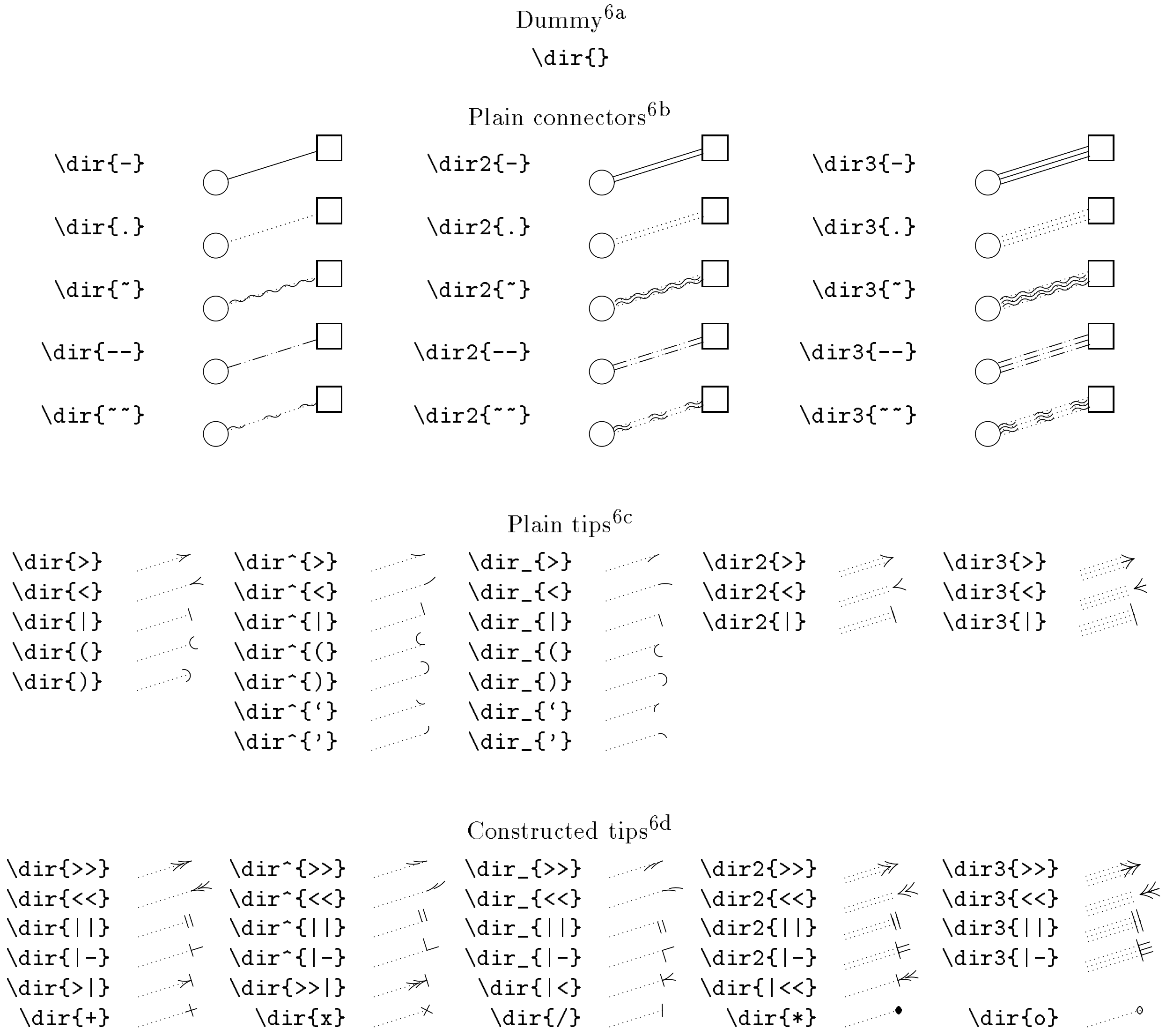


Figure 5: Kernel library ⟨dir⟩ectionals

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.

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`

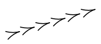
---

that will switch the use of such off and on.

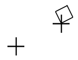
As can be seen by the last two columns, these (and most of the other connectors) also exist in double and triple versions with a `2` or a `3` prepended to the name. For convenience `\dir{=}` and `\dir{:}` are synonyms for `\dir2{-}` and `\dir2{.}`, respectively; similarly `\dir{==}` is a synonym for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



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



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

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 C&&c&& &&&&&&&&

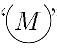
Circle ⟨object⟩s are round and typeset a segment of the circle centered at the reference point. The syntax of circles is described in figure 6 with explanations below.

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 4k) 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 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 option interface

Note: LaTeX users should also consult the paragraph on "xy.sty" in §1.1.

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 load the Xy-pic option file `xy`⟨option⟩`.tex`; `\xyrequire` will do so only if it is not already loaded, if it is then nothing happens.

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 afterwords.

Finally a description of the format of option files: they must look like

Syntax			Action
`\cir` ⟨radius⟩ `{` ⟨cir⟩ `}`			⟨cir⟩cle segment with ⟨radius⟩
⟨radius⟩	⟶	⟨empty⟩	use R_c as the radius
	|	⟨vector⟩	use X of the ⟨vector⟩ as radius
⟨cir⟩	⟶	⟨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⟩	⟶	`^`	anticlockwise
	|	`_`	clockwise

Figure 6: ⟨cir⟩cles.

```
%% ⟨identification⟩
%% ⟨copyright, ...⟩

\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 6 characters chosen among the lowercase letters (`a`–`z`), digits (`0`–`9`), and dash (`-`).

⟨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.

Part II

E

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:

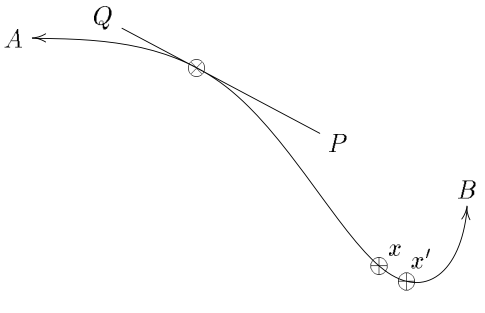
```
\xy (0,20)*+{A};(60,0)*+{B}
**\crv{}
**\crv{(30,30)}
**\crv{(20,40)&(40,40)}
**\crv{(10,20)&(30,20)&(50,-20)&(60,-10)}
\endxy
```

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) finds 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.

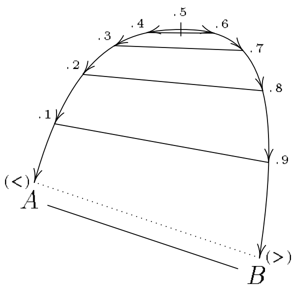
Using `?` will set the current direction to be tangential at that ⟨place⟩, and one can ⟨slide⟩ specified distances along the curve from a found ⟨place⟩ using the `?`. . . `/`⟨dimen⟩`/` notation:



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.)

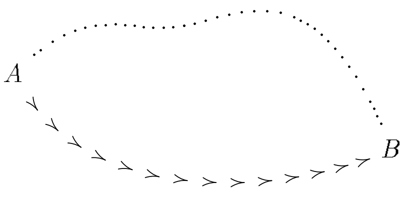
To have the same ⟨pos⟩ occuring as a multiple control point simply use a delimiter, which leaves the ⟨pos⟩ unchanged. Thus `\curve{`⟨pos⟩`&}` uses a cubic spline, whereas `\curve{`⟨pos⟩`}` is quadratic.

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.

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}
```

Syntax			Action
`\curve`⟨modifier⟩`{`⟨curve-object⟩⟨poslist⟩`}`			construct curved connection
⟨modifier⟩	⟶	⟨empty⟩	zero or more modifiers possible; default is `~C`
	|	`~`⟨curve-option⟩ ⟨modifier⟩	set ⟨curve-option⟩
⟨curve-option⟩	⟶	`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⟩	⟶	⟨empty⟩	use the appropriate default style
	|	`~*`⟨object⟩ ⟨curve-object⟩	specify the "drop" object[8a] and maybe more[8c]
	|	`~**`⟨object⟩ ⟨curve-object⟩	specify the "connect" object[8b] and maybe more[8c]
⟨poslist⟩	⟶	⟨empty⟩ | ⟨pos⟩ ⟨delim⟩ ⟨poslist⟩	list of positions for the control points
	|	`~@` | `~@` ⟨delim⟩ ⟨poslist⟩	add the current stack[8h] to the control points
⟨delim⟩	⟶	`&`	allowable delimiter

Figure 7: Syntax for curves.

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

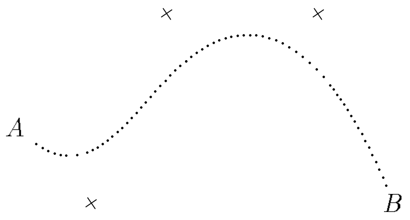
When there is no "connect" object then the tangent calculations are not carried out, resulting in a saving of time and memory; this is the default behaviour.

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.

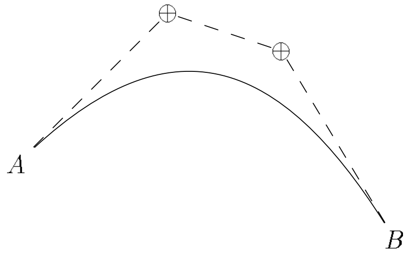
8f. By default the control points are marked with a small cross, specified by `*\dir{x}`. The "connect" object is ignored completely.



was typeset by . . .

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

8g. With lines joining 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⟩.

Note: Curves will be accessible to users through a `\crv`⟨dir⟩ command that makes a curve out of every directional. This is not finished yet.

9 Frame and Bracket extension

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.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`⟨modifiers⟩`{`⟨frame⟩`}`

---

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

Below we distinguish between ordinary frames and ‘brackets’.

9.1 F&&&&&

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

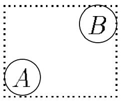
Notes

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. Two frames put just rules in the picture: `\frm{,}` puts a shade beneath the (assumed rectangular) object giving the illusion of ‘lifting’ it; `\frm<`⟨dimen⟩`>{,}` makes this shade ⟨dimen⟩ deep. `\frm{*}` just puts a black rule on top of the object.

`\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}`?

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.

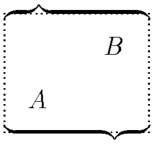
9.2 B&&c&&&&

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

Notes

9e. *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?



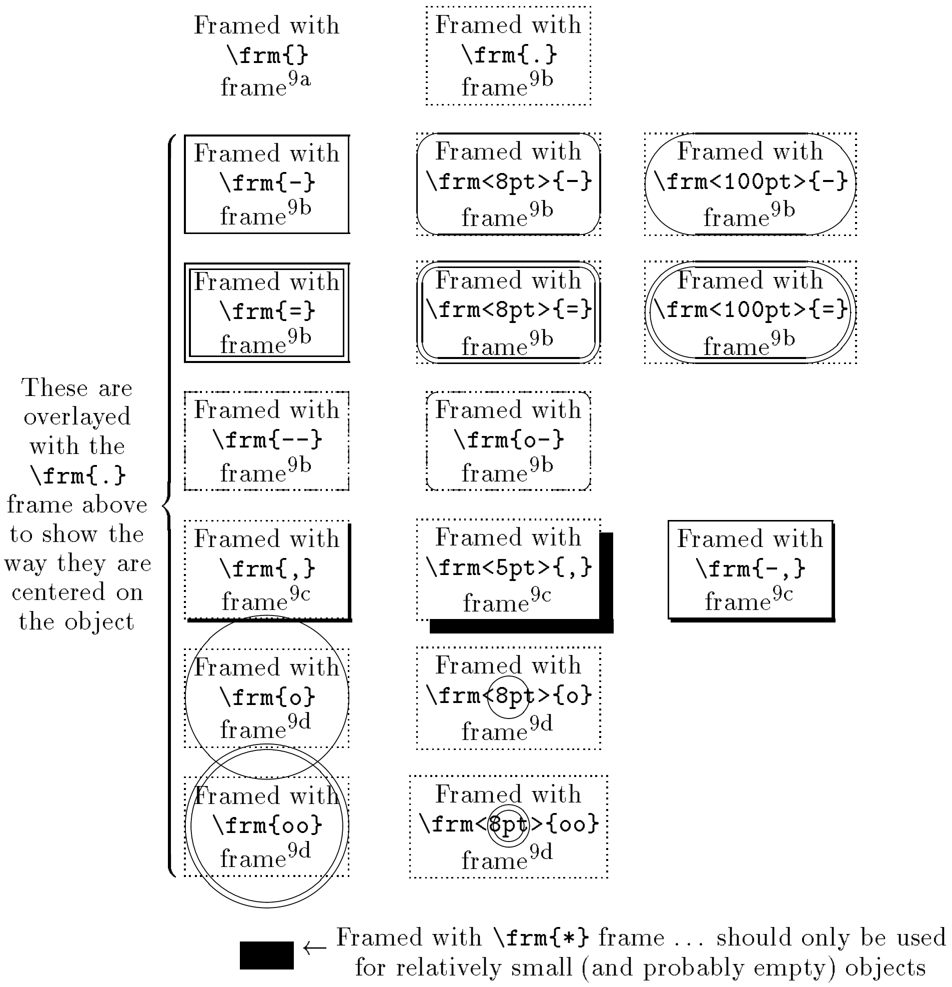


Figure 8: Plain ⟨frame⟩s.

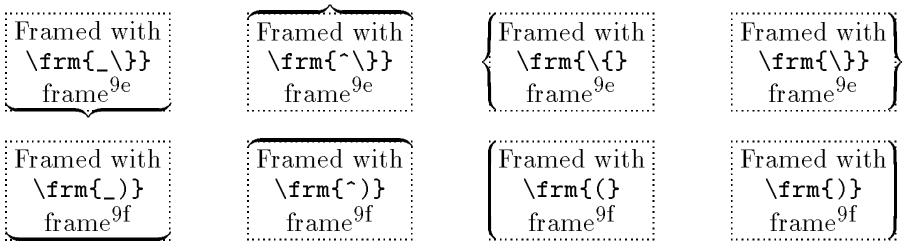


Figure 9: Bracket ⟨frame⟩s.

9f. *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 requires that the computer modern `cmex` font is loaded in font position 3.

To Do: Some new frames and several new brackes should be added.

10 Computer Modern tip extension

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{cmtip}`

This option provides arrow heads in the style of the Computer Modern fonts by Knuth (see [6] and [5, appendix F]). These are often more pleasing in connection with curved arrows.

The user can switch the "computer modern" versions of the directionals shown in figure 10 on and off with these declarations:

```
\UseComputerModernTips
\NoComputerModernTips
```

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

```
\xy*{} \ar
 @{*{\UseComputerModernTips\dir{<}}%
   -*{\NoComputerModernTips\dir{>}}}
 (20,5)*{} \endxy
```

will typeset

regardless of the tip choice in the surrounding text.

11 Line styles extension

Vers. 2.12 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

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 an argument.

`[thicker]`	double line thickness
`[thinner]`	halve line thickness
`[|`⟨dimen⟩`]`	set thickness to ⟨dimen⟩
`[|=`⟨word⟩`]`	make `[`⟨word⟩`]` set current style settings
`[|*]`	reuse previous style

Later settings of the linewidth override earlier settings; multiple calls to `[thicker]` and `[thinner]` compound.

Saving styles Once specified for an ⟨object⟩, the collection of styles can be assigned a name, via `[|=`⟨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. A warning message will be placed in the log file:

```
XY-pic Warning: Defining new style [⟨word⟩]
```

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` or `\diagram`. 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.

Colours This extension supports a few standard colours as styles: `[red]`, `[green]`, `[blue]`, `[cyan]`, `[magenta]`, `[yellow]`, `[black]`, `[white]` and `[gray]`. More extensive colour support is available using the `color` extension.

The diagram in figure 11, page 24, uses different line-thicknesses and colours.

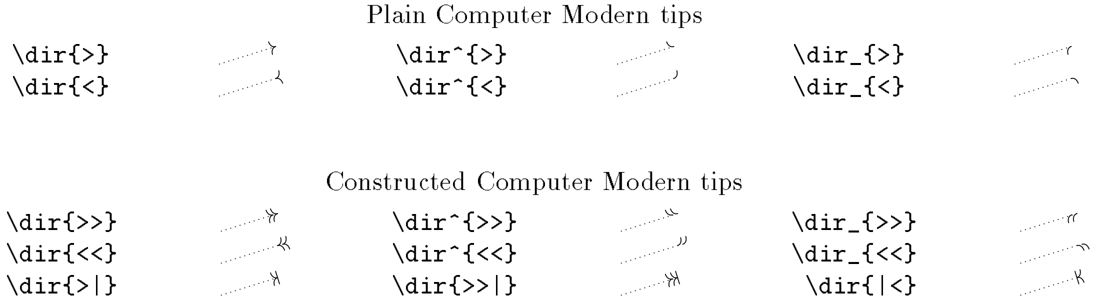


Figure 10: Computer Modern ⟨dir⟩ectionals

12 Rotate and Scale extension

Vers. 2.12 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 11 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 11.

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.

13 Colour extension

Vers. 2.10 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.

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.

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 new colour
`\UseCrayolaColors`	extra 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 $0 \leq r, g, b, c, m, y, k, s \leq 1$. 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 §11, then the current colour setting (if any) is saved also. Subsequent use of `[`⟨word⟩`]` recovers the colour and accompanying line-style settings.

Further colour names are defined by the command `\UseCrayolaColours` that loads the file `xyps-col.tex` where more colours are defined (consult the file for the colours and their their specifications in the RGB or CMYK models):

xyps-col.tex: This included file (version 2.10) provides definitions for the 68 colours recognised by name by Tomas Rokicki's `dvips` driver [10]. These colours become available for use in XY-pic pictures and diagrams, as `[`⟨shape⟩`]` modifiers.

The information has been copied from Rokicki's `color.pro` PostScript prolog file: "There are 68 predefined colours, with names taken primarily from the Crayola crayon box of 64 colours" [10, §16.1].

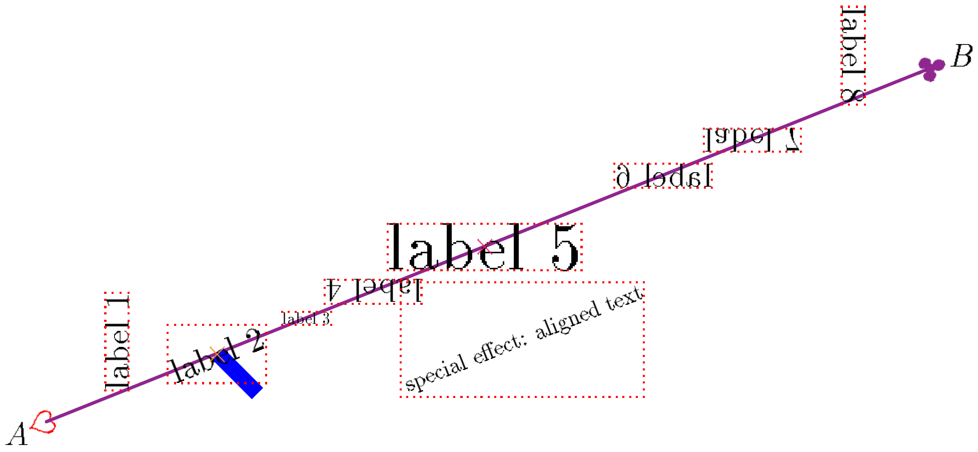


Figure 11: Rotations, scalings and flips

Part III

ea

Syntax			Action
`\PATH` ⟨path⟩			interpret ⟨path⟩
`\afterPATH{`⟨decor⟩`}` ⟨path⟩			interpret ⟨path⟩ and then run ⟨decor⟩
⟨path⟩	⟶	`~` ⟨action⟩ `{` ⟨stuff⟩ `}` ⟨path⟩	set ⟨action⟩[16a] to ⟨stuff⟩
	|	`~ + {` ⟨labels⟩ `}` ⟨path⟩	set default ⟨labels⟩[16b]
	|	`~ {` ⟨stuff⟩ `}` ⟨path⟩	set failure continuation[16c] to ⟨stuff⟩
	|	`'` ⟨segment⟩ ⟨path⟩	make straight segment[16d]
	|	`` ` `` ⟨turn⟩ ⟨segment⟩ ⟨path⟩	make turning segment[16f]
	|	⟨segment⟩	make last segment[16g]
⟨turn⟩	⟶	⟨diag⟩ ⟨turnradius⟩	1/4 turn[16f] starting in ⟨diag⟩
	|	⟨cir⟩ ⟨turnradius⟩	explicit turn[16f]
⟨turnradius⟩	⟶	⟨empty⟩	use default turn radius
	|	`/` ⟨dimen⟩	set *turnradius* to ⟨dimen⟩
⟨segment⟩	⟶	⟨path-pos⟩ ⟨slide⟩ ⟨labels⟩	segment[16e] with ⟨slide⟩ and ⟨labels⟩
⟨slide⟩	⟶	⟨empty⟩ | `<` ⟨dimen⟩ `>`	optional slide[16h]: ⟨dimen⟩ in the "above" direction
⟨labels⟩	⟶	`^` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	label with ⟨it⟩[16i] *above* ⟨anchor⟩
	|	`_` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	label with ⟨it⟩[16i] *below* ⟨anchor⟩
	|	`|` ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩	break with ⟨it⟩[16j] 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[16k]
	|	`*` ⟨object⟩	⟨it⟩ is an ⟨object⟩
	|	`@` ⟨dir⟩	⟨it⟩ is a ⟨dir⟩ectional
⟨alias⟩	⟶	⟨empty⟩ | `="`⟨id⟩`"`	optional name for label object[16l]

Figure 12: ⟨path⟩s

The `=<>` actions are always expanded in that sequence after *p* and *c* have been set up to the proper start and end of the segment but *before* any ⟨labels⟩ are interpreted, the `-` action is expanded for each subsegment *after* all ⟨labels⟩ have been interpreted (see also note 16d), and finally the `/` action is applied.

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 §16.2 below is little more than a special `\PATHaction` command and a clever defaulting mechanism.

16b. Defining default ⟨labels⟩ will insert these first in the label sequence of every ⟨segment⟩. 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
 ~={**{}}
 ~>{\save?>*\dir{>}\restore}
 ~-{**\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.

16c. 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.

16d. 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.
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 eith the `~+`-defined ones) are interpreted and typeset as described below.
4. Finally the *subsegment actions* are expanded: If there were n breaks then there are $n+1$ subsegments and thus `\PATHaction-{` ⟨stuff⟩ `}` will be expanded $n+1$ times. The ith expansion, $i \in \{1, \ldots, n+1\}$, will be performed with

$$\begin{array}{rcl} p & = & b_0 \;\texttt{.}\; b_{i-1} \\ c & = & b_{n+1} \;\texttt{.}\; b_i \end{array}$$

   where b_i denotes break i except that b_0 is the start and b_{n+1} the end object of the segment.

Example: Typically `~=` is used to do something that will setup the `?`⟨place⟩ format to suit the segment connection which is then used by `~<` to add something to the 'tail' of the path and by `~>` to add to its 'head', and finally `~-` is used to actually typeset the connection beween the given breaks. For example,

```
\xy*+{0}\PATH
 ~={**i\dir{-}}
 ~<{\save;?<*\dir{|}\restore}
 ~>{\save?>*\dir{>}\restore}
 ~-{**\dir{-}}
 '(10,1)*+{1}|b '(20,-2)*+{2} (30,0)*+{3}
\endxy
```

will build a 'mapsto path'

as follows: For each segment we do the following: (1) let `=` typeset an *invisible* connection that will make `?` behave correctly; (2) let `<` make the start point (p) of the first segment be a `\dir{|}` on the edge of the original p (the `;`s make us modify p rather than c); (3) let `>` make the end point of the last segment be a `\dir{>}` tip; and (4) let `-` typeset each subsegment of the connection as a solid line (that will trace the invisible one set up in (1)).

Numerous variations are possible by varying what goes in which actions, *e.g.*,

```
 ~={**i\dir{-}
    \save;?<*\dir{|}; ?>*\dir{>}
    \restore}
 ~-{**\dir{-}}
```

typesets

with every segment a separate mapsto arrow, and

```
 ~={**i\dir{-}}
 ~-{**\dir{-}
    \save;?<*\dir{|}; ?>*\dir{>}
    \restore}
```

typesets

16e. 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`.

16f. 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.

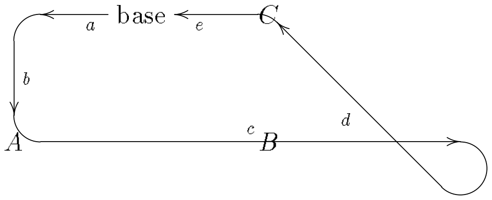
What the turn looks like is determined by the ⟨turn⟩ form:

⟨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

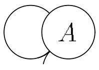
---

`\turnradius` ⟨add op⟩ `{`⟨dimen⟩`}`

---

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

Exercise 23: Typeset



using ⟨turn⟩s.

16g. The last segment is exactly as a straight one except that the `>` action (if any) is executed (and cleared) just after the `<` action.

16h. “Sliding” a segment means moving each of the p, c objects in the direction perpendicular to the current direction at each.

16i. 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 16d 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.

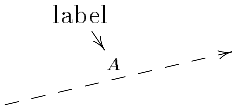
16j. Breaking means to “slice a hole” in the connection and insert ⟨it⟩ there. This is realized by typesetting the connection in question in *subsegments*, one leading to the break and one continuing after the break as described in notes 16a and 16d.

16k. 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.

16l. 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



Syntax			Action
`\ar` ⟨arrow⟩ ⟨path⟩			make ⟨arrow⟩ along ⟨path⟩
⟨arrow⟩	⟶	⟨form⟩*	⟨arrow⟩ has the ⟨form⟩s
⟨form⟩	⟶	@ ⟨variant⟩	use ⟨variant⟩ of arrow
	|	@ ⟨variant⟩ { ⟨tip⟩ }	build arrow[16m] using ⟨variant⟩ of a standard stem and ⟨tip⟩ for the head
	|	@ ⟨variant⟩ { ⟨tip⟩ ⟨conn⟩ ⟨tip⟩ }	build arrow[16m] using ⟨variant⟩ of ⟨tip⟩, ⟨conn⟩, and other ⟨tip⟩ as arrow tail, stem, and head (in that order)
	|	@/ ⟨direction⟩ ⟨dist⟩ /	curve[16o] arrow the ⟨dist⟩ance towards ⟨direction⟩
	|	@' { ⟨control points⟩ }	curve arrow using control points[16p]
	|	@* { ⟨modifier⟩* }	use object ⟨modifier⟩s[16q] for all objects
	|	| ⟨anchor⟩ ⟨it⟩	break each segment at ⟨anchor⟩ with ⟨it⟩
	|	^ ⟨anchor⟩ ⟨it⟩ | _ ⟨anchor⟩ ⟨it⟩	label each segment at ⟨anchor⟩ with ⟨it⟩
⟨variant⟩	⟶	⟨empty⟩ | ^ | _ | 0 | 1 | 2 | 3	⟨variant⟩: plain, above, below, double, or triple
⟨tip⟩	⟶	⟨tipchar⟩*	directional named as the sequence of ⟨tipchar⟩s
	|	⟨dir⟩	any ⟨dir⟩ectional[16n]
⟨tipchar⟩	⟶	< | > | (|) | | | ' | ` | + | /	recognised tip characters
	|	⟨letter⟩ | ⟨space⟩	more tip characters
⟨conn⟩	⟶	⟨connchar⟩*	directional named as the sequence of ⟨connchar⟩s
	|	⟨dir⟩	any ⟨dir⟩ectional[16n]
⟨connchar⟩	⟶	- | . | ~ | = | :	recognised connector characters

Figure 13: ⟨arrow⟩s.

16.2 A&&&&&

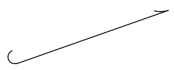
Arrows are paths with a particularly easy syntax for setting up arrows with *tail*, *stem*, and *head* in the style of [12]. This is provided by a single ⟨decor⟩ation the syntax of which is described in figure 13 (with the added convention that a raised '*' means 0 or more repetitions of the preceeding nonterminal).

Notes

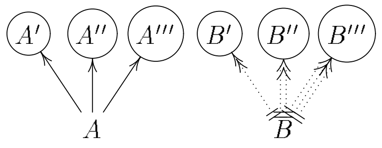
16m. Building an ⟨arrow⟩ is simply using the specified directionals (using `\dir` of §6.1) to build a path: the first ⟨tip⟩ becomes the *arrow tail* of the arrow, the ⟨conn⟩ection in the middle becomes the *arrow stem*, and the second ⟨tip⟩ becomes the *arrow head*. If a ⟨variant⟩ is given before the `{` then that variant `\dir` is used for all three. For example,

```
\xy\ar @^{(->} (20,7)\endxy
```

typesets



Exercise 25: Typeset these arrows:



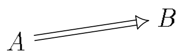
The above is a flexible scheme when used in conjunction with the kernel `\newdir` to define all sorts of arrowheads and -tails. For example,

```
\newdir{|>}{!/4.5pt/\dir{|}
            *:(1,-.2)\dir^{>}
            *:(1,+.2)\dir_{>}}
```

defines a new arrow tip that makes

```
\xy (0,0)*+{A}
 \ar @{=|>} (20,3)*+{B}
\endxy
```

typeset

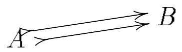


Notice that the fact that the directional uses only ⟨tipchar⟩ characters means that it blends naturally with the existing tips.

Exercise 26: Often tips used as 'tails' have their ink on the wrong side of the point where they are placed. Fortunately space is also a ⟨tipchar⟩ so we can define `\dir{ >}` to generate a 'tail' arrow. Do this such that

```
\xy (0,0)*+{A}="a", (20,3)*+{B}="b"
 \ar @{>->}  "a";"b" < 2pt>
 \ar @{ >->} "a";"b" <-2pt>
\endxy
```

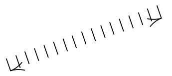
typesets



16n. Specifying a ⟨dir⟩ as a ⟨tip⟩ or ⟨conn⟩ means that `\dir`⟨dir⟩ is used for that ⟨tip⟩ or ⟨conn⟩. For example,

```
\xy\ar @{<^{|}>} (20,7)\endxy
```

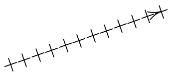
typesets



When using this you must specify a `{}` dummy ⟨dir⟩ectional in order to ignore one of the tail, stem, or tip components, *e.g.*,

```
\xy\ar @{{}{+}>} (20,7)\endxy
```

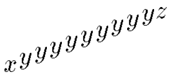
typesets



In particular `*`⟨object⟩ is a ⟨dir⟩ so any ⟨object⟩ can be used for either of the tail, stem, or head component:

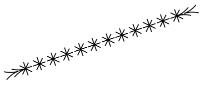
```
\xy\ar @{*{x}*{y}*{z}} (20,7)\endxy
```

typesets



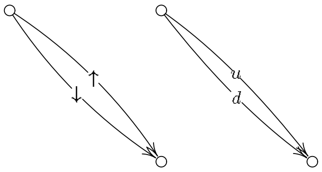
Note: A `*` introduces an ⟨object⟩ whereas the directional '•' is typeset by the ⟨dir⟩ `{*}`.

Exercise 27: Typeset



using only one `\ar` command.

16o. *Curving* the arrow by $/d\ell/$, where d is a ⟨direction⟩ and ℓ a ⟨dimen⟩sion, makes the stem a curve which is similar to a straight line but has had it's center point 'dragged' the distance ℓ in d:



was typeset by

```
\xy
 \POS (0,10)  *\cir<2pt>{} ="a"
    , (20,-10)*\cir<2pt>{} ="b"
 \POS"a" \ar @/^1ex/ "b"|\uparrow
 \POS"a" \ar @/_1ex/ "b"|\downarrow
%
 \POS (20,10) *\cir<2pt>{} ="a"
    , (40,-10)*\cir<2pt>{} ="b"
 \POS"a" \ar @/u1ex/ "b"|u
 \POS"a" \ar @/d1ex/ "b"|d
\endxy
```

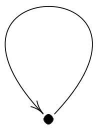
This is really just a shorthand for curving using the more general form described next: `@/`$d\ell$`/` is the same as `@'{ @+{**{} ?+/`$d\ 2\ell$` /} }` which makes the (quadratic) curve pass through the point defined by the ⟨pos⟩ `**{} ?+/`$d\ell$`/`.

16p. The second curve form is the more general one where more than one control point can be defined. The kernel stack is used for this purpose: the ⟨control points⟩ should be a ⟨pos⟩ pushing the control points in sequence on the stack: with the sequence $c_1, \ldots, c_k$ of control ⟨coord⟩inates this results in the ⟨form⟩

`@'{ @+`$c_1 \ldots$`@+`c_k`}`

See the curve extension described in §8 for the way the control points are used.

Exercise 28: Typeset the 'balloon arrow'



Hint: it uses a curve with three control points.

16q. A `@*{...}` formation defines what object ⟨modifier⟩s should be used when building objects that are part of the arrow. This is mostly useful in conjunction with extensions that define additional `[`⟨shape⟩`]` modifiers, *e.g.*, if a `[red]` ⟨modifier⟩ changes the colour of an object to red then `@*{[red]}` will make the entire arrow red.

All the features of ⟨path⟩s described above are available for arrows.

17 Two-cell feature

Vers. 2.12 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{2cell}`

This feature is designed to facilitate the typesetting of curved arrows, either singly or in pairs, together with labels on each part and between. The intended mathematical usage is for typesetting categorical "2-cell" morphisms and "pasting diagrams", for which special features are provided. These features also allow attractive non-mathematical effects.

The 2-cell feature makes use of facilities from the 'curve' extension which is therefore automatically loaded.

17.1 &&p&&&&&&&&& 2-c&&&&& && T&&p&&&&&&&

Categorical "2-cell" morphisms are used in the study of tensor categories and elsewhere. The morphisms are displayed as a pair of curved arrows, symmetrically placed, together with an orientation indicated by a short broad arrow, or *Arrow*. Labels may be placed on all three components.

$$\begin{tikzcd}
A \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] \arrow[r, phantom, "\Downarrow"] & B
\end{tikzcd}$$

```
\diagram
A\rtwocell^f_g &B\\
\enddiagram
```

$$\begin{tikzcd}
A \arrow[r, bend left=50, "f"] \arrow[r, "g"' near start] \arrow[r, bend right=50, "h"'] \arrow[r, phantom, shift left=3, "\Downarrow\alpha"] \arrow[r, phantom, shift right=3, "\Downarrow\beta"] & B
\end{tikzcd}$$

```
\diagram
A\ruppertwocell^f{\alpha}
  \rlowertwocell_h{\beta}
  \rto_(.35)g & B\\
\enddiagram
```

These categorical diagrams frequently having a matrix-like layout, as with commutative diagrams. To facilitate this there are control sequences of the form: `\rtwocell` , `\ultwocell` , `\xtwocell` , ... analogous to the names defined in `xyv2` for use in diagrams produced using `xymatrix`. As this involves the definition of 21 new control sequences, many of which may never be used, these are not defined immediately upon loading `xy2cell`. Instead the user must first specify `\UseTwocells`.

As in the second example above, just the upper or lower curved arrow may be set using control sequences of the form `\..uppertwocell` and `\..lowertwocell`. These together with the `\..compositemap` family, in which two abutting arrows are set with an empty object at the join, allow for the construction of complicated "pasting diagrams" (see figure 14 for an example).

The following initialise the families of control sequences for use in matrix diagrams.

`\UseTwocells`	two curves
`\UseHalfTwocells`	one curve
`\UseCompositeMaps`	2 arrows, end-to-end
`\UseAllTwocells`	(all the above)

Alternatively 2-cells can be set directly in XY-pictures without using the matrix feature. In this case the above commands are not needed. This is described in §17.5.

Furthermore a new directional `\dir{=>}` can be used to place an "Arrow" anywhere in a picture, after the direction has been established appropriately. It is used with all of the 2-cell types.

Labels are placed labels on the upper and lower arrows, more correctly 'anti-clockwise' and 'clockwise', using `^` and `_`. These are entirely optional with the following token, or grouping, giving the contents of the label. When used with `\..compositemap` the `^` and `_` specify labels for the first and second arrows, respectively.

Normally the label is balanced text, set in TEX's math mode, with `\twocellstyle` setting the style. The default definition is given by ...

```
\def\twocellstyle{\scriptstyle}
```

This can be altered using `\def` in versions of TEX or `\redefine` in LATEX. However labels are not restricted to being simply text boxes. Any effect obtainable using the XY-pic kernel language can be set within an `\xybox` and used as a label.

The position of a label can be altered by *nudging* (see below). Although it is possible to specify multiple labels, only the last usage of each of `^` and `_` is actually set, previous specifications being ignored.

Similarly a label for the central Arrow must be given, after the other labels, by enclosing it within braces `{...}`. An empty group `{}` gives an empty label; this is necessary to avoid misinterpretation of subsequent tokens.

17.2 &&&&&&&&& &p&&&&&&

The orientation of the central Arrow may be reversed, turned into an equality, or omitted altogether. In each case a label may still be specified, so in effect the Arrow may be replaced by anything at all.

These effects are specified by the first token in the central label, which thus has the form: `{`⟨tok⟩*label*`}` where ⟨tok⟩ may be one of ...

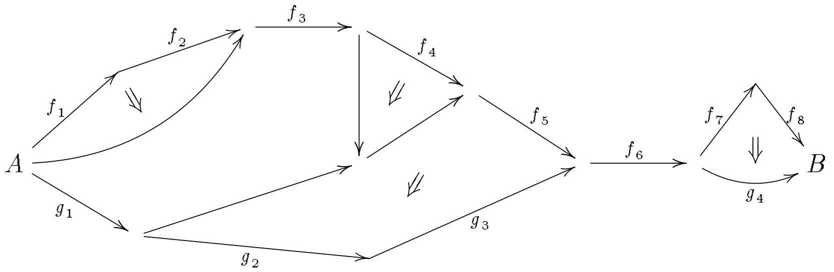


Figure 14: Pasting diagram.

`_`	Arrow points clockwise
`^`	Arrow points anti-clockwise
`=`	no tip, denotes equality
`\omit`	no Arrow at all.

When none of these occurs then the default of `_` is assumed. If the label itself starts with one of these characters then specify `_` explicitly, or enclose the label within a group `{...}`. See *Extra Options 1*, for more values of ⟨tok⟩.

17.3 &&&&&&&

Positions of all labels may be adjusted, as can the amount of curvature for the curved arrows. The way this is done is by specifying a "nudge" factor ⟨num⟩ at the beginning of the label. Here ⟨num⟩ is a number which specifies the actual position of the label in units of `\xydashl@` (the length of a single dash, normally 5pt) except with `\..compositemap`, see below. Movement is constrained to the perpendicular bisector of the line $\overline{cp}$. When nudging the label for the central Arrow it is the whole Arrow which is moved, along with its label.

Curvature of the arrows themselves is altered by a nudge of the form `\..twocell`⟨num⟩.... The separation of the arrows, along the bisector, is set to be ⟨num⟩`\xydashl@`. When ⟨num⟩ is zero, that is `\..twocell<0>...`, the result is a single straight arrow, its mid-point being the origin for nudging labels. A negative value for ⟨num⟩ is also acceptable; but check the orientation on the Arrow and which of `^` and `_` correspond to which component.

The origin for nudging labels is where the arrow crosses the bisector. Positive nudges move the label outwards while negative nudges move towards $\overline{pc}$ and possibly beyond. The default position of a label is on the outside, with edge at the origin.

The origin for nudging the Arrow is at the midpoint of $\overline{pc}$. A positive nudge moves in the clockwise direction. This will be the direction of the arrowhead, unless it has been reversed using `^`.

Labels on a `\..compositemap` are placed relative to the midpoint of the component arrows. Nudges are in units of 1pt. Movement is in the usual XY-pic *above* and *below* directions, such that a positive nudge is always outside the triangle formed by the arrows and line $\overline{pc}$.

The special nudge value `<\omit>` typesets just the Arrow, omitting the curved arrows entirely. When used with labels, the nudge value `<\omit>` causes the following label to be ignored.

Exercise 29: Give code to typeset figure 14.
Such code is relatively straight-forward, using "nudging" and `\omit` to help position the arrows, curves and Arrows. It also uses an *excursion*, as described below in the subsection *Extra Options 3*.

17.4 E&&&& &$_p$&&&&&

The following features are useful in non-mathematical applications.

1. no Arrow

This is determined by special values for ⟨tok⟩ as the first (or only) character in the central label, as in the above description of the standard options.

`'`	arrowheads pointing clockwise;
`` ` ``	arrowheads pointing anti-clockwise;
`"`	arrow tips on both ends;
`!`	no tips at all.

The central Arrow is omitted, leaving symmetrically placed curved connections with arrowheads at the specified ends. A label can be placed where the Arrow would have been.

If a special arrowhead is specified using `~'{..}` (see Extra Options 2, below) then this will be used instead of the standard `\dir{>}`.

$$\begin{tikzcd}
\text{Clouds} \arrow[r, bend left, "\text{precipitation}"] \arrow[r, phantom, "H_2O"] & \text{Oceans} \arrow[l, bend left, "\text{evaporation}"]
\end{tikzcd}$$

<table>
<tr><th>Syntax</th><th></th><th></th><th>Action</th></tr>
<tr><td>⟨twocell⟩</td><td>⟶</td><td>⟨2-cell⟩⟨options⟩⟨Arrow⟩</td><td>typeset ⟨2-cell⟩ with the ⟨options⟩ and ⟨Arrow⟩</td></tr>
<tr><td>⟨2-cell⟩</td><td>⟶</td><td>\..twocell</td><td>typeset two curved arrows</td></tr>
<tr><td></td><td>|</td><td>\..uppertwocell</td><td>typeset upper curved arrow only</td></tr>
<tr><td></td><td>|</td><td>\..lowertwocell</td><td>typeset lower curved arrow only</td></tr>
<tr><td></td><td>|</td><td>\..compositemap</td><td>use consecutive straight arrows</td></tr>
<tr><td>⟨Arrow⟩</td><td>⟶</td><td>{⟨tok⟩⟨text⟩}</td><td>specifies orientation and label</td></tr>
<tr><td></td><td>|</td><td>{⟨nudge⟩⟨text⟩}</td><td>adjust position, use default orientation</td></tr>
<tr><td></td><td>|</td><td>{⟨text⟩}</td><td>use default position and orientation</td></tr>
<tr><td>⟨tok⟩</td><td>⟶</td><td>^ | _ | =</td><td>oriented anti-/clockwise/equality</td></tr>
<tr><td></td><td>|</td><td>\omit</td><td>no Arrow, default is clockwise</td></tr>
<tr><td></td><td>|</td><td>` | ' | " | !</td><td>no Arrow; tips on two curved arrows as:
anti-/clockwise/double-headed/none</td></tr>
<tr><td>⟨options⟩</td><td>⟶</td><td>⟨option⟩⟨options⟩</td><td>list of optional modifications</td></tr>
<tr><td>⟨option⟩</td><td>⟶</td><td>⟨empty⟩</td><td>use defaults</td></tr>
<tr><td></td><td>|</td><td>^ ⟨label⟩</td><td>place ⟨label⟩ on the upper arrow</td></tr>
<tr><td></td><td>|</td><td>_ ⟨label⟩</td><td>place ⟨label⟩ on the lower arrow</td></tr>
<tr><td></td><td>|</td><td>⟨nudge⟩</td><td>set the curvature, based on ⟨nudge⟩ value</td></tr>
<tr><td></td><td>|</td><td>\omit</td><td>do not set the curved arrows</td></tr>
<tr><td></td><td>|</td><td>!</td><td>place \modmapobject midway along arrows</td></tr>
<tr><td></td><td>|</td><td>~ ⟨what⟩ { ⟨object⟩ }</td><td>use ⟨object⟩ in place specified by ⟨what⟩</td></tr>
<tr><td>⟨what⟩</td><td>⟶</td><td>⟨empty⟩</td><td>set curves using the specified ⟨object⟩</td></tr>
<tr><td></td><td>|</td><td>^ | _</td><td>use ⟨object⟩ with upper/lower curve</td></tr>
<tr><td></td><td>|</td><td>` | '</td><td>use ⟨object⟩ for arrow head/tail</td></tr>
<tr><td>⟨label⟩</td><td>⟶</td><td>⟨text⟩ | ⟨nudge⟩ ⟨text⟩</td><td>set ⟨text⟩ displaced by ⟨nudge⟩</td></tr>
<tr><td>⟨nudge⟩</td><td>⟶</td><td><⟨number⟩></td><td>positions object along a fixed axis</td></tr>
<tr><td></td><td>|</td><td><\omit></td><td>do not typeset the object</td></tr>
</table>

Figure 15: ⟨twocell⟩s

```
\xymatrixcolsep{5pc}
\diagram
 \relax\txt{Clouds }\rtwocell<10>
 _{\hbox{\tiny evaporation }}
 ^{\hbox{\tiny precipitation }}
{'{\boldmath{H_2 O}}}
&\relax\txt{Oceans}\\
\enddiagram
```

$$\begin{tikzcd}
\text{Mathematics} \arrow[r, bend left, leftrightarrow, "\text{theory}"] \arrow[r, bend right, leftrightarrow, "\text{experiment}"'] & \text{Physics}
\end{tikzcd}$$

```
\xymatrixcolsep{5pc}
\diagram
\relax\txt{\llap{Math}ematics }\rtwocell
_{\hbox{\tiny experiment }}
^{\hbox{\tiny theory }}{"}
& \relax\txt{Physics} \\
\enddiagram
```

2. Changing Tips and Module Maps

The following commands are provided for specifying the ⟨object⟩ to be used when typesetting various parts of the twocells.

command	*default*
`\modmapobject{`⟨object⟩`}`	`\dir{|}`
`\twocellhead{`⟨object⟩`}`	`\dir{>}`
`\twocelltail{`⟨object⟩`}`	`\dir{}`
`arrowobject{`⟨object⟩`}`	`\dir{=>}`
`\curveobject{`⟨object⟩`}`	
`\uppercurveobject{`⟨object⟩`}`	`{}`
`\lowercurveobject{`⟨object⟩`}`	`{}`

These commands set the object to be used for all subsequent 2-cells at the same level of TEX grouping. `\curveobject` specifies both of the upper- and lower-curve objects. For some of these there is also a way to change the object for the current 2-cell only. This requires a `~`-⟨option⟩ which is described below, except for the `\..curveobject` types, which are discussed in *Extra Options 4*.

These effects are specified by placing options after the `\..twocell` control sequence, *e.g.* `\rtwocell`*options labels*... . Each option is either a single token ⟨tok⟩, or a `~`⟨tok⟩ with a single argument: `~`⟨tok⟩`{`*arg*`}`. Possibilities are listed in the following table, in which `{..}` denotes the need for an argument.

`\omit`	no arrows, Arrow and label only;
`!`	place module-map indicator;
`~'{..}`	change arrow-head to `{..}`;
``~`{..}``	place/change tail on arrow(s);
`~{..}`	change object used to set curves;
`~^{..}`	use object `{..}` to set upper curve;
`~_{..}`	use object `{..}` to set lower curve;

Here we discuss the use of `!`, `~'`, ``~` `` and `\omit`. The description of `~^`, `~_` and `~{..}` is given in *Extra Options 4*.

The default module map indicator places a single dash crossing the arrow at right-angles, located roughly midway along the actual printed portion of the arrow, whether curved or straight. This takes into account the sizes of the objects being connected, thereby giving an aesthetic result when these sizes differ markedly. This also works with `\..compositemap` where an indicator is placed on each arrow. The actual object can be changed using `\modmapobject`.

Any of the standard XY-pic tips may be used for arrow-heads. This is done using `~'{..}`, for example `~'{\dir{>>}}` gives double-headed arrows. Similarly ``~`{..}`` can be used to place an arrow-tail. Normally the arrow-tail is , so is not placed; but if a non-empty tail has been specified then it will be placed, using `\drop`. No guarantee is offered for the desired result being obtained when an arrow-tail is mixed with the features of *Extra Options 1*.

$$\begin{tikzcd}
P \arrow[r, bend left, twoheadrightarrow, "M", "\otimes" marking] \arrow[r, bend right, twoheadrightarrow, "M'"', "\otimes" marking] \arrow[r, phantom, "f \Vert"] & S
\end{tikzcd}$$

```
\modmapobject{\objectbox{\otimes}}
\xymatrixcolsep{5pc}
\diagram
P\rtwocell~!~'{\dir{>>}}~`{\dir{|}}
 ^{<1.5>M}_{<1.5>M'}{=f} & S \\
\enddiagram
```

3. Excursions

The syntax for the `\x..twocell` types and for `\xcompositemap` is a little different to what might be expected from that for `\xto`, `\xline`, etc. For example,

`\xtwocell[`⟨hop⟩`]{`⟨displace⟩`}...`

connects to the ⟨pos⟩ displaced by ⟨displace⟩ from the relative cell location specified by ⟨hop⟩. The displacement can be any string of valid XY-pic commands, but they must be enclosed within a group `{...}`. When the cell location is required, a null grouping `{}` *must* be given.

When used with the `<\omit>` nudge, such excursions allow a labelled Arrow to be placed anywhere within an XY-pic diagram; furthermore the Arrow can be oriented to point in any direction.

4. Fancy curves

By specifying `\curveobject` an arbitrary object may be used to construct the curved arrows. Indeed with a `\..twocell` different objects can be used with the upper and lower curves by specifying `\uppercurveobject` and `\lowercurveobject`.

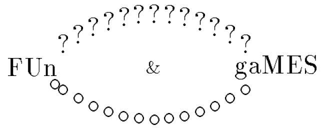
These specifications apply to all 2-cells subsequently constructed at the same level of TEX grouping. Alternatively using a `~`-option, as in *Extra Options 2*, allows such a specification for a single 2-cell or curved part.

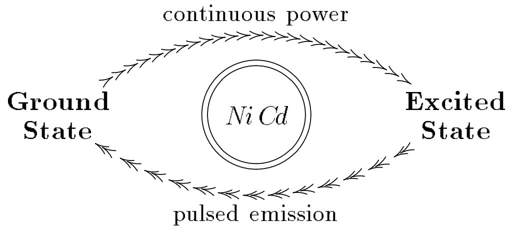
Objects used to construct curves can be of two types. Either a single ⟨object⟩ is set once, with copies placed along the curve. Alternatively a directional object can be aligned with the tangent along the curve. In this case use a specification takes the form:

`\curveobject{`⟨spacer⟩`~**`⟨object⟩`}`.

Here ⟨spacer⟩ may be any ⟨object⟩ of non-zero size. Typically it is empty space, *e.g.* `+`⟨dimen⟩`{}`.

Exercise 30: Give code to typeset the following diagrams.





17.5 2-c&&&& && &&&&-&&&&&&&&&&

Two-cells can also be set directly within any XY-picture, without the matrix feature, using either `\drop` or `\connect`.

```
\def\myPOS#1{\POS}\def\goVia#1{%
  \afterPOS{\connect#1\myPOS}}
\xy
 *+{A}="A",+<1cm,1.5cm>*+{B}="B",
 +<2.0cm,0pt>*+{C}="C",
 +<1cm,-1.5cm>*+{D}="D",
"A";\goVia{\uppertwocell^\alpha{}}"B"{}
;\goVia{\twocell^\zeta_\xi{\gamma}}"C"{}
;\goVia{\compositemap{}}"D"{},
"A";\goVia{\lowertwocell{}}"D"{}
\endxy
```

$$\begin{tikzcd}
 & B \arrow[rr, bend left, "\zeta"] \arrow[rr, bend right, "\xi"'] \arrow[rr, phantom, "\Downarrow\gamma"] & & C \arrow[dr] & \\
A \arrow[ur, bend left, "\alpha"] \arrow[rrrr, bend right] \arrow[rrrr, phantom, "\Downarrow"] \arrow[urr, phantom, "\Searrow"] & & & \arrow[ur, phantom, "\Swarrow"] & D
\end{tikzcd}$$

The code shown is a compact way to place a chain of 2-cells within a picture. It illustrates a standard technique for using `\afterPOS` to find a ⟨pos⟩ to be used for part of a picture, then subsequently reuse it. Also it is possible to use `\drop` or ⟨decor⟩s to specify the 2-cells, giving the same picture.

```
\xy *+{A}="A",+<1cm,1.5cm>*+{B}="B",
 +<2cm,0pt>*+{C}="C",
 +<1cm,-1.5cm>*+{D}="D",
 "A";"B"\uppertwocell^\alpha{}
 \POS"B";"C"
 \twocell^\zeta_\xi{\gamma}\POS"C";
 \afterPOS{\drop\compositemap{}}"D"
 {}\POS "A";
 \afterPOS{\drop\lowertwocell{}}"D"
\endxy
```

The `\connect` variant is usually preferable as this maintains the size of the object at c, while the `\drop` variant leaves a rectangular object having p and c on opposite sides.

18 Matrix feature

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{matrix}`

This option implements "XY-matrices", *i.e.*, matrices where it is possible to refer to the entry objects by their row/column address. We first describe the general form of XY-matrices in §18.1, then in §18.2 we summarise the new ⟨coord⟩inate forms used to refer to entries. In §18.3 we explain what parameters can be set to change the spacing and orientation of the matrix, and in §18.4 we explain how the appearance of the entries can be changed.

18.1 &&-&&&&&c&&

The fundamental command of this feature is the command `\xymatrix{...}` that reads a matrix of entries in the generic TEX row&column format, *i.e.*, where rows are separated with `\\` and contain columns separated with `&`. Thus a matrix with *maxrow* rows and *maxcol* columns where each entry contains *row*,*col* is entered as

```
\xymatrix{
 1,1 &        1,2 &   ···   1,maxcol \\
 2,1 &        2,2 &         2,maxcol \\
 ⋮                 ⋱
 maxrow,1 &  maxrow,2 &  maxrow,maxcol }
```

(TEXnically the `&` character represents any 'alignment tab', *i.e.*, character with category code 4).

A ⟨matrix⟩ can appear either in an XY-picture (as ⟨decor⟩) or "stand-alone".

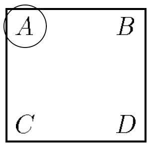
The points where `\xymatrix` is different from ordinary matrix constructions (like plain TEX's `\matrix{...}` and LATEX's `array` environment) are

- arbitrary XY-pic ⟨decor⟩ations may be specified in each entry and will be interpreted in a state where *c* is the current entry,
- the entire matrix is an object itself with reference point as the top left entry, and
- a progress message "`<xymatrix` *rows*`x`*cols* *size*`>`" is printed for each matrix with *rows* × *cols* entries and XY-pic complexity *size* (the number of primitive operations performed).
- Entries starting with a `*` are special (described in §18.4)[6], so use `{*}` to get a `*`.

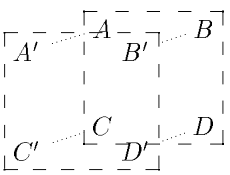
For example,

```
$$\xy
 \xymatrix{A&B\\C&D}
 \drop\frm{-}
 \drop\cir<8pt>{}
\endxy$$
```

will typeset



In fact entries of one matrix may refer to entries of another by using the ⟨pos⟩ save mechanism:



was typeset (using the 'frame' extension) by

```
$$\xy
 \xymatrix {
  A\POS="A" & B\POS="B" \\
  C\POS="C" & D\POS="D" }
 \POS*\frm{--}
 \POS-(10,3)
 \xymatrix {
  A'\POS;"A"**\dir{.}
    & B'\POS;"B"**\dir{.} \\
  C'\POS;"C"**\dir{.}
    & D'\POS;"D"**\dir{.} }
 \POS*\frm{--}
\endxy$$
```

Bug: Matrices cannot be nested.

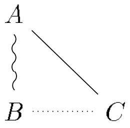
18.2 &&& $_c$&&&&&&&&& &&&&&&&

it is possible within entries to refer to all the entries of the XY-matrix using the following special ⟨coord⟩inate forms:

`"`r`,`c`"`	Position and extents of entry in row r, column c (top left is `"1,1"`)
`[`Δr`,`Δc`]`	Δr rows below and Δc columns right of current entry
`[` ⟨hop⟩* `]`	entry reached by the ⟨hop⟩s; each ⟨hop⟩ is one of `dulr` describing one 'move' to a neighbor entry

So the current entry has the synonyms `[0,0]`, `[]`, `[rl]`, `[ud]`, `[dudu]`, etc.

These forms are useful for defining diagrams where the entries are related, *e.g.*,



was typeset by

```
$$\xy
\xymatrix{
 A \POS[];[d]**\dir{~},
       [];[dr]**\dir{-}    \\
 B & C \POS[];[l]**\dir{.} }
\endxy$$
```

If an entry outside the XY-matrix is referenced then an error is reported.

18.3 &$_p$&$_c$&&& &&& &&&&&&&&

The default spacing distances between rows and columns are called *rowsep* and *colsep*. They can be

---

[6] In general it is recommended that entries start with a non-expanding token, *i.e.*, an ordinary (non-active) character, `{`, or `\relax`.

changed from the default `2pc` by two special commands similar to the ones for the defaults in the kernel:

`\xymatrixrowsep` ⟨add op⟩ `{`⟨dimen⟩`}`
`\xymatrixcolsep` ⟨add op⟩ `{`⟨dimen⟩`}`

The spacing around each object can also be changed through modifiers as explained in the following section.

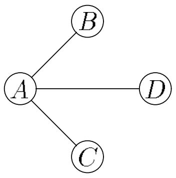
An entire matrix can be rotated by adding a *rotation prefix* between the `\xymatrix` command and the opening `{`:

`@`⟨direction⟩

This will set the orientation of the rows to ⟨direction⟩ (the default corresponds to `@r`).

Note: Rotation is experimental and the spacing of a rotated matrix may change in future versions.

Exercise 31: How did the author typeset the following matrix?



Hint: It is a 2 × 2 matrix and the author used `\entrymodifiers = {[o]}` and `\everyentry = {\drop\cir{}}` as explained in the next section.

18.4 E&&&&&&

The object ⟨modifier⟩s used for the default entries can be changed from the default '`!C +=<`*objectwidth*`,` *objectheight*`> +<`2 × *objectmargin*`>`' (with the effect of centering the object, forcing it to have at least the size *objectwidth* times *objectheight* and finally add the *objectmargin*) to all sides, by

`\entrymodifiers={` ⟨stuff⟩ `}`

The appearance of a single entry can be modified by entering it as

`*` ⟨object⟩ ⟨decor⟩

This makes the particular entry ignore the entry modifiers and typeset as a kernel object with the same reference point as the (center of) the default object would have had.

Exercise 32: Typeset the following diagram:

$$\begin{tikzcd}
A \times B \arrow[r, "/A"] \arrow[d, "/B"'] & B \arrow[d, "\times A"] \\
A \arrow[r, "B\times"'] & B \times A
\end{tikzcd}$$

Finally, `\everyentry` is used to setup ⟨decor⟩ that should be inserted before everything else in each entry. Initially it is empty but

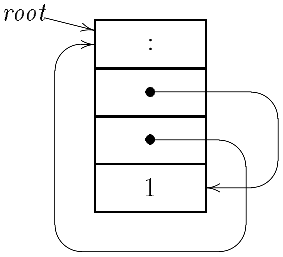
`\everyentry={` ⟨decor⟩ `}`

will insert ⟨decor⟩ first in each entry. For example,

```
\everyentry={\drop\cir{}}
\xy\xymatrix{
 A \POS[];[r]**\dir{~} & B
}\endxy
```

will typeset

Exercise 33: How did the author typeset the following diagram?



Hints: The arrow feature was used to make the bending arrows and the frame extension for the frames around each cell.

19 Graph Combinator feature

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{graph}`

This option implements 'XY-graph', a special *combinatoric drawing language* suitable for diagrams like flow charts, directed graphs, and various forms of trees. The base of the language is reminiscent of the PIC [4] language because it uses a notion of the 'current location' and is based on 'moves'. But the central construction is a 'map' combinator that is borrowed from functional programming.

XY-graph make use of facilities of the 'arrow' feature option which is therefore required.

Figure 16 summarises the syntax of a ⟨graph⟩ with notes below. A ⟨graph⟩ can appear either in an XY-picture (as ⟨decor⟩) or "stand-alone".

Notes

19a. A *move* is to establish a new *current node*.

Syntax			Action
`\xygraph{`⟨graph⟩`}`			typeset ⟨graph⟩
⟨graph⟩	⟶	⟨step⟩*	interpret ⟨step⟩s in sequence
⟨step⟩	⟶	⟨node⟩	move^{19a} to the ⟨node⟩
	|	`-` ⟨node⟩ ⟨labels⟩	draw^{19b} line to ⟨node⟩, with ⟨labels⟩
	|	`:`⟨arrow⟩ ⟨node⟩ ⟨labels⟩	draw^{19b} ⟨arrow⟩ to ⟨node⟩, with ⟨labels⟩
	|	`(` ⟨list⟩ `)`	map^{19c} current node over ⟨list⟩
⟨node⟩	⟶	`[` ⟨move⟩ `]`	new node ⟨move⟩d relative to current
	|	`"`⟨id⟩`"`	previously saved^{19d} node
	|	`?`	currently mapped^{19c} node
	|	`!` ⟨escape⟩	interpret material in another mode
	|	⟨node⟩ ⟨it⟩	⟨node⟩ with ⟨it⟩ typeset and saved^{19d} there
	|	⟨node⟩ `= "`⟨id⟩`"`	⟨node⟩ saved^{19d} as `"`⟨id⟩`"`
⟨move⟩	⟶	⟨hop⟩*	⟨hop⟩s^{19e} (`dulr`) from current node
⟨list⟩	⟶	⟨graph⟩ `,` ⟨list⟩ | ⟨graph⟩	list of subgraphs^{19c}
⟨escape⟩	⟶	`{` ⟨pos⟩ ⟨decor⟩ `}`	perform ⟨pos⟩ ⟨decor⟩^{19f}
	|	`M` ⟨matrix⟩	insert ⟨matrix⟩^{19g}
	|	`P` ⟨matrix⟩	insert ⟨polygon⟩^{19i}

Figure 16: ⟨graph⟩s

19b. To *draw* something is simply to draw a line or the specified ⟨arrow⟩ from the current node to the specified target node. The target then becomes the current node. All the features of arrows as described in §16 can be used, in particular arrows can be labelled and segmented, but with the change that ⟨path-pos⟩ means ⟨node⟩ as explained in note §16e.

19c. To *map over a list* is simply to save the current node and then interpret the ⟨list⟩ with the following convention:

- Start each element of the list with the current node as saved and p as the previous list element, and
- let the `?` ⟨node⟩ refer to the saved current node explicitly.

19d. Typeset ⟨it⟩ and make it the current node. Also saves ⟨it⟩ for later reference using `"`⟨id⟩`"`: if ⟨it⟩ is a simple letter, or digit, then just as `"`⟨it⟩`"`; if ⟨it⟩ is of the form `{`*text*`}` or `*`...`{`*text*`}` then as `"`*text*`"`.

With the `=` addition it is possible to save explicitly in case several nodes have the same text or a node has a text that it is impractical to use for reference.

Exercise 34: How did the author typeset this?

$$\begin{tikzcd} A \arrow[r, bend left] & A \arrow[r, bend left] & A \arrow[ll, bend left] \end{tikzcd}$$

19e. Moving by a series of *hops* is simply moving in a grid as the sequence of `dulr` (for down/up/left/right) indicates. The grid is a standard cartesian coordinate system with 3pc unit unless a base `"graphbase"` is defined or the current base is redefined using `!` with an appropriate ⟨pos⟩ition using `:` and `::` as described in note 3d.

To Do: Many more moves should be allowed, in particular these should be available: (1) 'until perpendicular to ...' and (2) 'until intercepts with ...'.

19f. This 'escapes' into the XY-pic kernel language and interprets the ⟨pos⟩ ⟨decor⟩. The current node is then set to the resulting c object and the grid from the resulting *base*.

The effect of the ⟨pos⟩ ⟨decor⟩ can be completely hidden from XY-graph by entering it as `{\save`...`\restore}`.

19g. **Note:** This only works when the 'matrix' feature has also been loaded. It inserts a node consisting of the ⟨matrix⟩ which must have the usual form (see §18 for the details):

```
⟨rotation⟩ { ⟨rows and columns⟩ }
```

Within the matrix the following two control sequences are specially defined: `\:` is defined as an alias for `\ar` and `\="`⟨id⟩`"` will save the entry as `"`⟨id⟩`"` (`\everyentry` is used for these).

Finally the grid is set as the top left 'square' of the matrix, *i.e.*, with `[d]` and `[r]` adjusted as they work in the top left entry (so `[dr]` immediately after the matrix will work as expected, *e.g.*, make the center of `"2,2"` the current node, but others might not, *e.g.*, `[rr]` will not necessarily place the current node on top of `"1,3"`.

19h. **Note:** This only works when the 'polygon' feature has also been loaded. It inserts a node consisting of the ⟨polygon⟩ which must have the usual form (see §20 for the details).

19i. It is possible to insert a ⟨polygon⟩ in a graph provided the `poly` option described in §20 has been loaded: it will have its center on top of the current node and default radius as the ⟨hop⟩ base size.

The canonical diagram example illustrates most of the above:

```
\xygraph{
 !M{ X \times_Z Y \="xy" \:[r]_p \:[d]^q
     & X \="X" \:[d]_f   \\
     Y \="Y" \:[r]^g & Z }
 [ul]U   ( ? :@/^.5pc/ ^x        "X" ,
   ? :@{-->}   |-{(x,y)} "xy" ,
   ? :@/_.5pc/ _y        "Y" )   }
```

typesets

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

20 Polygon feature

Vers. 2.12 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{poly}`

This feature provides a means for specifying the locations of vertices for regular polygons, with 3 to 12 sides. Polygons can be easily drawn and/or the vertex positions used to constuct complex graphics within an XY-picture. Many non-regular polygons can be specified by setting a non-square basis.

A polygon is most easily specified using ...

```
\xypolygon⟨number⟩{}     with ⟨number⟩ sides;
\xypolygon⟨number⟩{⟨tok⟩}    ⟨tok⟩ at vertices;
\xypolygon⟨number⟩{⟨object⟩}
      with a general ⟨object⟩ at each vertex;
```

Here ⟨number⟩ is a sequence of digits, giving the number of sides. If used within an `\xy`...`\endxy` environment then the polygon will be centred on c, the current ⟨pos⟩. However an `\xypolygon` can be used outside such an environment, as "stand-alone" polygon; the whole picture must be specified within the `\xypolygon` command.

In either case the shape is obtained by spacing vertices equally around the "unit circle" with respect to the current basis. If this basis is non-square then the vertices will lie on an ellipse. Normally the polygon, with at most 12 vertices, is oriented so as to have a flat base when specified using a standard square basis. With more than 12 vertices the orientation is such that the line from the centre to the first vertex is horizontal, pointing to the right. Any other desired orientation can be obtained, with any number of vertices, by using the `~={...}` as described below.

The general form for `\xypolygon` is ...

```
\xypolygon⟨number⟩"⟨prefix⟩"{⟨switches⟩...}
```

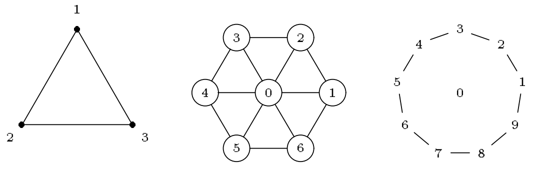
where the `"`⟨prefix⟩`"` and ⟨switches⟩ are optional. Their uses will be described shortly.

A `\xypolygon` establishes positions for the vertices of a polygon. At the same time various things may be typeset, according to the specified ⟨switches⟩. An ⟨object⟩ may be dropped at each vertex, "spokes" drawn to the centre and successive vertices may be connected as the polygon's "sides". Labels and breaks can be specified along the spokes and sides.

Each vertex is automatically named: `"1"`, `"2"`, ..., `"`⟨number⟩`"` with `"0"` as centre. When a ⟨prefix⟩ has been given, names `"`⟨prefix⟩`0"`, ..., `"`⟨prefix⟩⟨number⟩`"` are used instead. While the polygon is being constructed the macro `\xypolynum` expands to the number of sides, while `\xypolynode` expands to the number of each vertex, spoke and side at the time it is processed. This occurs in the following order: *vertex* 1, *spoke* 1, *vertex* 2, *spoke* 2, *side* 1, *vertex* 3, *spoke* 3, *side* 2, ..., *vertex* n, *spoke* n, *side* $n-1$, *side* n where the final side joins the last vertex to the first.

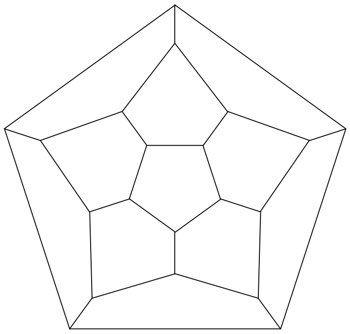
The macro `\xypolyname` holds the name of the polygon, which is ⟨prefix⟩ if supplied. In this case the value of `\xypolynum` is also stored as `\`⟨prefix⟩`NUMSIDES`, accessible outside the polygon.

As stated above, a polygon with up to 12 vertices is oriented so as to have a flat base, when drawn using a standard square basis. Its vertices are numbered in anti-clockwise order, commencing with the one at horizontal-right of centre, or the smallest angle above this (see example below). With more than 12 vertices then vertex `"1"` is located on the horizontal, extending to the right from centre (assuming a standard square basis). By providing a switch of the form `~={`⟨angle⟩`}` then the vertex `"1"` will be located on the unit circle at ⟨angle⟩° anti-clockwise from "horizontal" — more correctly, from the X-direction in the basis to be used when setting the polygon, which may be established using a `~:{...}` switch.



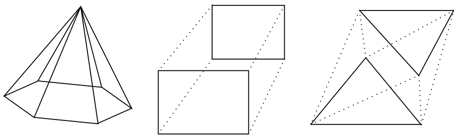
Exercise 35: Give code to typeset these.

One important use of ⟨prefix⟩ is to allow the vertices of more than one polygon to be accessed subsequently within the same picture. Here are some examples of this, incorporating the `~:{...}` switch to perform simple rescalings. Firstly the edges of a dodecahedron as a planar graph:



```
\xy /l1.5pc/:,{\xypolygon5"A"{}},
{\xypolygon5"B"{~:{(1.875,0):}~>{}}},
{\xypolygon5"C"{~:{(-2.95,0):}~>{}}},
{\xypolygon5"D"{~:{(-3.75,0):}}},
{"A1"\PATH~/{**\dir{-}}'"B1"'"C4"'"B2"},
{"A2"\PATH~/{**\dir{-}}'"B2"'"C5"'"B3"},
{"A3"\PATH~/{**\dir{-}}'"B3"'"C1"'"B4"},
{"A4"\PATH~/{**\dir{-}}'"B4"'"C2"'"B5"},
{"A5"\PATH~/{**\dir{-}}'"B5"'"C3"'"B1"},
"C1";"D1"**\dir{-},"C2";"D2"**\dir{-},
"C3";"D3"**\dir{-},"C4";"D4"**\dir{-},
"C5";"D5"**\dir{-} \endxy
```

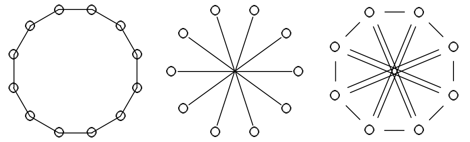
Next a hexagonal pyramid, a rectangular box and an octahedral crystal specified as a triangular anti-prism. Notice how the `~:{...}` switch is used to create non-square bases, allowing the illusion of 3D-perspective in the resulting diagrams:



```
\xy/r2pc/: ="A", +(.2,1.5)="B","A",
{\xypolygon6{~:{(1,-.1):(0,.33)::}
 ~<>{;"B"**\dir{-}}}}\endxy
\quad \xy /r2pc/:
{\xypolygon4"A"{~:{(0,.7)::}}},+(.7,1.1),
{\xypolygon4"B"{~:{(.8,0):(0,.75)::}}},
"A1";"B1"**\dir{.},"A2";"B2"**\dir{.},
"A3";"B3"**\dir{.},"A4";"B4"**\dir{.}
\endxy\quad \xy /r2pc/:
{\xypolygon3"A"{~:{(0,.7)::}}},+(.7,1.1),
{\xypolygon3"B"{~:{(-.85,0):(-.15,.8)::}}}
,"A1"\PATH~/{**\dir{.}}'"B2"'"A3"'"B1"
'"A2"'"B3"'"A1" \endxy
```

Vertex object: Unless the first character is `~`, signifying a "switch", then the whole of the braced material is taken as specifying the ⟨object⟩ for each vertex. It will be typeset with a circular edge using `\drop[o]...`, except when there is just a single token ⟨tok⟩. In this case it is dropped as `\drop=0{`⟨tok⟩`}`, having zero size. An object can also be dropped at each vertex using the switch `~*{...}`, in which case it will be circular, with the current *objectmargin* applied.

The next example illustrates three different ways of specifying a `\circ` at the vertices.



```
\xy/r2pc/: {\xypolygon12{\circ}},
+/r5pc/,{\xypolygon10{~<{-}~>{}{\circ}}},
+/r5pc/,{\xypolygon8{~*{\circ}~<=}}\endxy
```

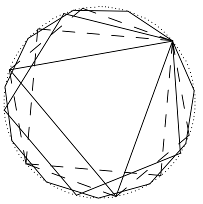
Switches

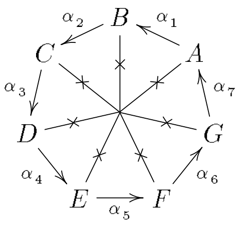
The allowable switches are given in the following table:

`~:{...}`	useful for rescaling.
`~*{`⟨object⟩`}`	⟨object⟩ at each vertex.
`~={`⟨angle⟩`}`	align first vertex.
`~<{...}`	directional for "spokes";
`~<<{`⟨arrow⟩`}`	use ⟨arrow⟩ for spokes;
`~<>{...}`	labels & breaks on spokes.
`~>{...}`	directional for "sides";
`~><{`⟨arrow⟩`}`	use ⟨arrow⟩ for sides;
`~>>{...}`	labels & breaks on sides.

Using `~<<{`⟨arrow⟩`}` or `~><{`⟨arrow⟩`}` is most appropriate when arrowheads are required on the sides or spokes, or when labels/breaks are required. Here ⟨arrow⟩ is as in figure 13, so it can be used simply to specify the style of directional to be used. Thus `~<<{}` sets each spoke as a default arrow, pointing outwards from the centre; `~<<{@{-}}` suppresses the arrowhead, while `~><{@{}}` uses an empty arrow along the sides. Labels and breaks are specified with `~<>{`...`}` and `~>>{`...`}`, where the `{`...`}` use the notation for a ⟨label⟩, as in figure 12.

When no tips or breaks are required then the switches `~<{`...`}` and `~>{`...`}` are somewhat faster, since less processing is needed. Labels can still be specified with `~<>{`...`}` and `~>>{`...`}`, but now using the kernel's ⟨place⟩ notation of figure 1. In fact any kernel code can be included using these switches. With `~<>` the current p and c are the centre and vertex respectively, while for `~>>` they are the current vertex and the previous vertex. (The connection from vertex `"`⟨number⟩`"` to vertex `"1"` is done last.) The pyramid above is an example of how this can be used. Both `~<{`...`}` and `~<<{`⟨arrow⟩`}` can be specified together, but only the last will actually be used; similarly for `~>{`...`}` and `~><{`⟨arrow⟩`}`.





```
\def\alphanum{\ifcase\xypolynode\or A
\or B\or C\or D\or E\or F\or G\or H\fi}
\xy/r3pc/: {\xypolygon3{~={40}}},
{\xypolygon4{~={40}~>{{--}}}},
{\xypolygon5{~={40}}},
{\xypolygon6{~={40}~>{{--}}}},
{\xypolygon11{~={40}}},
{\xypolygon50{~={40}~>.}}, +/r8pc/,
{\xypolygon7{~<<{@{-}}~><{}
 ~<>{|*\dir{x}}~*{\alphanum}
 ~>>{_{\alpha_\xypolynode^{}}}}}
\endxy
```

Use of the `~={`...`}` switch was described earlier. When using the `~:{`...`}` more can be done than just setting the base. In fact any kernel code can be supplied here. It is processed prior to any other part of the polygon. The graphics state has c at the centre of the polygon, p at the origin of coordinates within the picture and has basis unchanged from what has previously been established. The current point c will be reset to the centre following any code interpreted using this switch.

A further simplification exists for sides and spokes without ⟨arrow⟩s. If ⟨tok⟩ is a single character then `~>`⟨tok⟩, `~>{`⟨tok⟩`}`, `~>{{`⟨tok⟩`}}` all specify the directional `\dir{`⟨tok⟩`}`; similarly with the `~<` switch. On the other hand, compound directionals require all the braces, e.g. `~>{{--}}` and `~>{2{.}}`.

After all switches have been processed, remaining tokens are used to specify the ⟨object⟩ for each vertex. Such tokens will be used directly after a `\drop`, so can include object ⟨modifier⟩s as in figure 3. If an ⟨object⟩ has already been specified, using the `~*` switch, then the following message will be written to the TEX log:

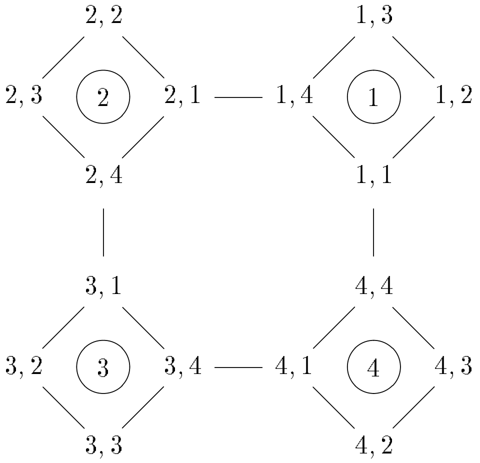
```
XY-pic Warning:  vertex already specified,
        discarding unused tokens:
```

with tokens at the end indicating what remains unprocessed. Similarly extra tokens before the `{`...`}` generate a message:

```
XY-pic Warning: discarding unused tokens:
```

Nested Polygons

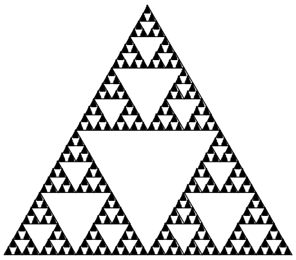
When `\xypolygon` is specified within a `~<>{`...`}` or `~>>{`...`}` switch for another polygon, then the inner polygon inherits a name which incorporates the number of the part on which it occurs, as given by `xypolynode`. This name is accessed using `\xypolyname`. In the following example the inner polygon is placed using `~<>` in order to easily adjust its orientation to the outward direction of the spokes.



```
\xypolygon4{~:{/r6pc/:}
 ~<>{*\frm<10pt>{o}\xypolygon4{~:{/-2.5pc/:}
   ~*{\xypolyname\xypolynode}}}
 [o]=<7pc>{\xypolynode}}
```

Notice how nested polygons inherit names `"1,1"`, `"1,2"`, ..., `"4,1"`, ..., `"4,4"` for their vertices. If a ⟨prefix⟩ is supplied at the outermost level then the names become: `"`⟨prefix⟩i, j`"`. Specifying a ⟨prefix⟩ for the inner polygon overrides this naming scheme. The same names may then be repeated for each of the inner polygons, allowing access afterwards only to the last—possibly useful as a memory saving feature when the vertices are not required subsequently.

Four levels of nesting gives a quite acceptable "Sierpinski gasket". The innermost triangle is provided by `\blacktriangle` from the AMS symbol font `msam5`, at 5-point size. Further levels can be achieved using the PostScript backend, otherwise line segments become too small to be rendered using XY-fonts.



```
\def\objectstyle{\scriptscriptstyle}
\xypolygon3{~:{/r5.2pc/:}
 ~>{}~<>{?\xypolygon3"a"{~:{(.5,0):}
 ~>{}~<>{?\xypolygon3"b"{~:{(.5,0):}
 ~>{}~<>{?\xypolygon3"c"{~:{(.5,0):}
 ~>{}~<>{?\xypolygon3"d"{~:{(.5,0):}
~<>{?*!/d.5pt/=0{\blacktriangle}}
}} }} }} }} }
```

Note the use of naming in this example; when processing this manual it saves 13,000+ words of main memory and 10,000+ string characters as well as 122 strings and 319 multi-letter control sequences.

21 Version 2 Compatibility feature

Vers. 2.12 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{v2}`

This option provides backwards compatibility with XY-pic version 2: diagrams written according to the "Typesetting diagrams with XY-pic: User's Manual" [13] should typeset correctly with this option loaded

There are a few exceptions: the features described in §21.1 below are not provided because they are not as useful as the author originally thought and thus virtually never used. And one extra command is provided to speed up typesetting of documents with XY-pic version 2 diagrams by allowing the new compilation functionality on old diagrams.

The remaining sections list all the obsolete commands and suggest ways to achieve the same things using XY-pic 2.12, *i.e.*, without the use of this option. They are grouped as to what part of XY-pic replaces them; the compilation command is described last.

Note: "version 2" is meant to cover all public releases of XY-pic in 1991 and 1992, *i.e.*, version 1.40 and versions 2.1 through 2.6. The published manual cited above (for version 2.6) is the reference in case of variations between these versions, and only things documented in that manual will be supported by this option!

21.1 U&&&pp&&&&& &&c&&p&&&&&&&&&&&

Here is a list of known incompatibilities with version 2 even when the `v2` option is loaded.

- Automatic 'shortening' of arrow tails by `|<<` break was a bug and has been 'fixed' so it does not work any more. Put a `|<\hole` break before it.
- The version 2.6 `*` position operator is not available. Use the `:` and `::` operators.
- Using $t_1;t_2:(x,y)$ as the target of an arrow command does not work. Enclose it in braces, *i.e.*, write
$$\{t_1;t_2:(x,y)\}$$
- The older `\pit`, `\apit`, and `\bpit` commands are not defined. Use `\dir{>}` (or `\tip`) with variants and rotation.
- The even older notation where an argument in braces to `\rto` and the others was automatically taken to be a 'tail' is not supported. Use the supported `|<`... notation.

If you do not use these features then your version 2 (and earlier) diagrams should typeset the same with this option loaded except that sometimes the spacing with version 2.12 is slightly different from that of version 2.6 which had some spacing bugs.

21.2 &&&&&&&& &&&&&& &&&&&&&&

The following things are added to the kernel by this option and described here: idioms, obsolete positions, obsolete connections, and obsolete objects. For each we show the suggested way of doing the same thing without this option:

Removed AMS-TeX idioms

Some idioms from AMS-TeX are no longer used by XY-pic: the definition commands `\define` and `\redefine`, and the size commands `\dsize`, `\tsize`, `\ssize`, and `\sssize`. Please use the commands recommended for your format—for plain TeX these are `\def` for the first two and `\displaystyle`, `\textstyle`, `\scriptstyle`, and `\scriptscriptstyle` for the rest. The `v2` option ensures that they are available anyway.

Version also 2 used the AMS-TeX `\text` and a (non-object) box construction `\Text` which are emulated—`\text` is only defined if not already defined, however, using the native one (of AMS-TeX or AMS-LaTeX or whatever) if possible. Please use the `\txt` object construction directly since it is more general and much more efficient!

Obsolete state

In version 2 the available state dimensions had different names: `\cL`, `\cR`, `\cH`, and `\cD` for `\Lc`, `\Rc`, `\Uc`, and `\Dc`. These are made synonyms for the new names.

Obsolete position manipulation

In version 2 many things were done using individual ⟨decor⟩ control sequences that are now done using ⟨pos⟩ operators.

Version 2 positioning	Replacement
`\go`⟨pos⟩	`\POS;p,`⟨pos⟩
`\aftergo{`⟨decor⟩`}`⟨pos⟩	`\afterPOS{`⟨decor⟩`};p,`⟨pos⟩
`\merge`	`\POS.p\relax`
`\swap`	`\POS;\relax`
`\Drop{`⟨text⟩`}`	`\drop+{`⟨text⟩`}`

Obsolete connections

These connections are now implemented using directionals.

Version 2 connection	Replacement
`\none`	`\connect h\dir{}`
`\solid`	`\connect h\dir{-}`
`\Solid`	`\connect h\dir2{-}`
`\Ssolid`	`\connect h\dir3{-}`
`\dashed`	`\connect h\dir{--}`
`\Dashed`	`\connect h\dir2{--}`
`\Ddashed`	`\connect h\dir3{--}`
`\dotted`	`\connect h\dir{.}`
`\Dotted`	`\connect h\dir2{.}`
`\Ddotted`	`\connect h\dir3{.}`
`\dottedwith{`⟨text⟩`}`	`\connect h{`⟨text⟩`}`

Note how the 'hidden' specifier `h` should be used because version 2 connections did not affect the size of diagrams.

Obsolete tips

These objects all have `\dir`-names now:

Version 2 tip	Replacement
`\notip`	`\dir{}`
`\stop`	`\dir{|}`
`\astop`	`\dir^{|}`
`\bstop`	`\dir_{|}`
`\tip`	`\dir{>}`
`\atip`	`\dir^{>}`
`\btip`	`\dir_{>}`
`\Tip`	`\dir2{>}`
`\aTip`	`\object=<5pt>:(32,-1)\dir^{>}`
`\bTip`	`\object=<5pt>:(32,+1)\dir_{>}`
`\Ttip`	`\dir3{>}`
`\ahook`	`\dir^{(}`
`\bhook`	`\dir_{(}`
`\aturn`	`\dir^{'}`
`\bturn`	`\dir_{'}`

The older commands `\pit`, `\apit`, and `\bpit`, are not provided.

Obsolete object constructions

The following object construction macros are made obsolete by the enriched ⟨object⟩ format:

Version 2 object	Replacement
`\rotate(`⟨factor⟩`)`⟨tip⟩	`\object:(`⟨factor⟩`,`⟨factor⟩`){`⟨tip⟩`}`
`\hole`	`\object+{}`
`\squash`⟨tip⟩	`\object=0{`⟨tip⟩`}`
`\grow`⟨tip⟩	`\object+{`⟨tip⟩`}`
`\grow<`⟨dimen⟩`>`⟨tip⟩	`\object+<`⟨dimen⟩`>{`⟨tip⟩`}`
`\squarify{`⟨text⟩`}`	`\object+={`⟨text⟩`}`
`\squarify<`⟨dimen⟩`>{`⟨text⟩`}`	`\object+=<`⟨dimen⟩`>{`⟨text⟩`}`

where rotation is done in a slightly different manner in version 2.12 (it was never accurate in version 2).

21.3 &&&&&&&&& &&&&&&&&&&& & &&&&&&

Version 2 had commutative diagram functionality corresponding to the `frames` extension and parts of the `matrix` and `arrow` features. These are therefore loaded and some extra definitions added to emulate commands that have disappeared.

Frames

The version 2 frame commands are emulated using the frame extension (as well as the `\dotframed`,

`\dashframed`, and `\rounddashframed` commands communicated to some users by electronic mail):

Version 2 object	Replacement
`\framed`	`\drop\frm{-}`
`\framed<`⟨dimen⟩`>`	`\drop\frm<`⟨dimen⟩`>{-}`
`\Framed`	`\drop\frm{=}`
`\Framed<`⟨dimen⟩`>`	`\drop\frm<`⟨dimen⟩`>{=}`
`\dotframed`	`\drop\frm{.}`
`\dashframed`	`\drop\frm{--}`
`\rounddashframed`	`\drop\frm{o-}`

Matrices

The `\diagram` ⟨rows⟩ `\enddiagram` command is provided as an alias for `\xy\xymatrix{` ⟨rows⟩ `}\endxy` centered in math mode and `\LaTeXdiagrams` changes it to use `\begin` … `\end` syntax. `v2` sets a special internal 'old matrix' flag such that trailing `\\` are ignored and entries starting with `*` are safe.

`\NoisyDiagrams` is ignored because the matrix feature always outputs progress messages.

Finally the version 2 `\spreaddiagramrows` and `\spreaddiagramcolumns` spacing commands are emulated using `\xymatrixrowsep` and `\xymatrixcolsep`:

Arrows

The main arrow commands of version 2 were the `\morphism` and `\definemorphism` commands that have been replaced by the `\ar` command.

`v2` provides them as well as uses them to define the version 2 commands `\xto`, `\xline`, `\xdashed`, `\xdotted`, `\xdouble`, and all the derived commands `\dto`, `\urto`, …; the `\arrow` commands of the β-releases of v3 is also provided.

Instead of commands like `\rrto` and `\uldouble` you should use the arrow feature replacements `\ar[rr]` and `\ar@{=}[ul]`.

The predefined turning solid arrows `\lltou`, …, `\tord` are defined as well; these are now easy to do with ⟨turn⟩s.

21.4 &&&&&&&& &&&&&&&

The v2 User's Manual says that you can load XY-pic with the command `\input xypic` and as a LaTeX 2.09 'style option' `[xypic]`. This is made synonymous with loading this option by the files `xypic.tex` and `xypic.sty` distributed with the `v2` option.

`xypic.tex`: This file (version 2.10) just loads the `v2` feature.

`xypic.sty`: Loads `xy.sty` and the `v2` feature.

21.5 C&&$_p$&&&&& &2-&&&&&&&&

In order to make it possible to use the new compilation features even on documents written with XY-pic v2, the following command has been added:

`\diagramcompileto{` ⟨name⟩ `}` … `\enddiagram`

which is like the ordinary diagram command except the result is compiled into a file ⟨name⟩`.xyc`. Note that compilation is not quite safe in all cases!

There is also the following command that switches on *automatic compilation* of all diagrams created with the v2 `\diagram` … `\enddiagram` command:

`\CompileAllDiagrams {` ⟨prefix⟩ `}`

will apply `\xycompileto{`⟨prefix⟩n`}{`…`}` to each diagram with n a sequence number starting from 1.

If for some reason a diagram does not work when compiled then replace the `\diagram` command with `\diagramnocompile` (or in case you are using the LaTeX form, `\begin{diagramnocompile}`), or use

`\NoCompileAllDiagrams`
`\ReCompileAllDiagrams`

where the last switches compilation back on.

Part IV

Backend

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 `\special`s for ⟨one particular⟩ driver

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

22.1 C&&&&&&& &&& ᴛ&&-&&&&&&

To activate the use of PostScript the user must specify one of the following command that selects the format of the `\special`s to be used:

`\UsePSspecials {`⟨driver⟩`}`	
`\NoPSspecials`	cancels PostScript
`\UsePSspecials`	restores PostScript

The `\UsePSspecials` initially causes a special driver file (see below) to be read. This file contains definitions which are specific to the particular ⟨driver⟩. Note that some drivers may not be able to support all of the PostScript effects that can be requested from within Xy-pic. When an unsupported effect is encountered, it is simply ignored. A message warning that the requested effect is unavailable will be produced unless too many such messages have already been issued.

Use of fonts is restored at any point by calling `\NoPSspecials`, after which use of PostScript is restored by using `\UsePSspecials`, without need of an argument. This allows PostScript to be turned on and off for individual diagrams, or for portions of a single diagram. Use of these commands obeys normal TeX scoping rules, so if `\NoPSspecials` or `\UsePSspecials` is specified within an environment, the previous setting will be restored upon leaving that environment.

For users of LaTeX 2_ε, and presumably LaTeX3 (when it becomes available), the driver type will be inherited from any corresponding PostScript option specified with the `\documentclass` command, see [3, page 317]. The implicit `\UsePSspecials` will be executed at the `\begin{document}` line; hence any `\NoPSspecials` must occur after this to be effective.

The following table, which mimics the one in the stated LaTeX 2_ε reference, describes current support for PostScript drivers: × denotes full support, for all the features the driver can handle; ? denotes that some features have not been tested, but may still work; – denotes no support as yet. Please note the spelling, which corresponds to the way the respective writers refer to their own products within their own documentation. Alternative combinations of upper- and lowercase letters are not guaranteed to work correctly.

⟨driver⟩	Description	Xy-ps
`dvips`	Tomas Rokicki's dvips	×
`Textures`	Blue Sky Research's Textures	×
`OzTeX`	Andrew Trevorrow's OzTeX	×
`ln`	Digital Corp. printers	–
`dvitops`	James Clark's dvitops	?
`emtex`	Eberhard Matte's em-TeX	–

Other DVI-drivers may already work if they use conventions similar to **`dvips`**, OzTeX or Textures. The TeXnical documentation [11] in the file `xyps.doc` contains instructions concerning how to make Xy-ps work with other drivers. To have another driver specifically supported it is only necessary to inform the author of its existence, how it handles `\specials`, and negotiate with him a means for testing/verifying the implementation.

It should be possible to change ⟨driver⟩ up until such time as a `\special` is actually used. This is to allow users to switch from a system default. This ability is new with version 2.9; any difficulties with this feature should be reported to the author

The following lists the ⟨driver⟩s available, including some experimental ones not mentioned above. The associated driver file is given in parentheses, along with any special considerations needed when using them.

`dvips` for dvips (xyps-dvi.tex): This included file (version 2.10) provides Xy-ps support for the dvips driver by Tomas Rokicki [10] (it has been tested with dvips version 5.55a).

`Textures` for Textures (xyps-txt.tex): This included file (version 2.10) provides Xy-ps support for the DVI driver of Textures.[8] for the Macintosh.[9]

`OzTeX` for OzTeX (xyps-oz.tex): This included file (version 2.10) provides Xy-ps support for the DVI driver of OzTeX by Andrew Trevorrow.[10]

Bug: Colour support is not complete (see `INSTALL.OzTeX`

Note: To use Xy-pic effectively with OzTeX requires changing several parameters. This is described in the file **`INSTALL.OzTeX`** of the Xy-pic distribution.

`dvitops` for `dvitops` (xyps-dto.tex): This included file (version 2.10) provides Xy-ps support for the **`dvitops`** DVI driver by James Clark.

Bug: This code has not been tested!

---

[8] Textures is a product of Blue Sky Research. Xy-ps has been tested on versions 1.5b and later; no guarantee is given for earlier versions.

[9] Macintosh is a trademark of Apple Computer Inc.

[10] OzTeX v1.7 is a shareware implementation of TeX for Macintosh available from many bulletin boards and ftp sites; v1.5 and earlier versions were freeware. Email contact: ⟨akt150@huxley.anu.edu.au⟩.

`dviwindo` for `dviwindo` (xyps-wdo.tex): This included file (version 2.10) provides Xy-ps support for the `dviwindo` DVI driver.

Bug: This code has not been tested!

`dvipub` for `dvipub` (xyps-pub.tex): This included file (version 2.10) provides Xy-ps support for the `dvipub` DVI driver.

Bug: This code has not been tested!

Information to improve the abilities of these drivers should be conveyed to the author. Printed technical documentation or software would be the most useful form, though e-mail concerning good experiences would also be helpful. ☺

22.2 &&& &&& 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.
- Straight 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.[11]
- 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.

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.
- The DVI files created using Xy-ps 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.

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 user has the following commands available...

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

The `\UsePSheader` command must be specified before `\UsePSspecials{`⟨driver⟩`}` is invoked. It allows the name of the dictionary file to be specified as the ⟨filename⟩. Normally it is sufficient to invoke `\UsePSheader{}`, which will use the default dictionary name of `xy212dict.ps`, referring to the current version of Xy-pic and Xy-ps.

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.

---

[11] 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.

22.3 PostScript &&$_c$&$_p$&

An extra ⟨shape⟩ modifier key allows arbitrary PostScript code to be applied to the current ⟨object⟩.

`[!`⟨postscript code⟩`]`	for special effects
`[psxy]`	stores current location.

Normally the ⟨postscript code⟩ will be a simple command to alter the PostScript graphics state: e.g. `[!1 0 0 setrgbcolor]` changes the colour used to render parts of the ⟨object⟩. Any number of such ⟨shape⟩ modifiers is allowable, however it is more efficient to combine them into a single modifier, whenever possible.

It is very important that braces `{` and `}` do not appear explicitly in any ⟨postscript code⟩, as this may upset the Xy-pic ⟨object⟩ parsing. However it is acceptable to have a control sequence name here, expanding into more intricate PostScript code. This will not be expanded until a later (safe) time.

Due to differences within the DVI-drivers, such simple PostScript commands need not affect every part of an ⟨object⟩. In particular the lines, curves and arrowheads generated by Xy-pic use a different mechanism, which should give the same result with all drivers. This involves redefining some PostScript procedures which are always read prior to rendering one of these objects. One simple way to specify a red line is as follows; the `xycolor` extension provides more sophisticated support for colour. The ⟨shape⟩ modifiers described in the previous section also use this mechanism, so should work correctly with all drivers.

```
\def\colorxy(#1){%
 /xycolor{#1 setrgbcolor}def}
...
\connect[!\colorxy(1 0 0)]\dir{-}
...
```

Note how the braces are inserted within the expansion of the control sequence `\colorxy`, which happens after parsing of the ⟨connection⟩. The following table shows which graphics parameters are treated in this way, their default settings, and the type of PostScript code needed to change them.

colour	`/xycolor{0 setgray}def`
line width	`/xywidth{.4 setlinewidth}def`
dashing	`/xydash{[] 0 setdash}def`
line-cap	`/xycap{1 setlinecap}def`
line-join	`/xyjoin{1 setlinejoin}def`

This feature is meant primarily for modifying the rendering of objects specified in TeX and Xy-pic, not for drawing new objects within PostScript. No guarantee can be given of the current location, or scale, which may be different with different drivers. However a good PostScript programmer will be able to overcome such difficulties and do much more. To aid in this the special modifier `[psxy]` is provided to record the location where the reference point of the current ⟨object⟩ will be placed. Its coordinates are stored with keys `xyXpos` and `xyYpos`.

22.4 E&&&&&&&&&&

Several included file handle standard extensions.

`xyps-l.tex`: This included file (version 2.9) provides Xy-ps support for the effects defined in the `line` extension.

`xyps-c.tex`: This included file (version 2.9) provides Xy-ps support for the effects defined in the `color` extension.

`xyps-r.tex`: This included file (version 2.9) provides Xy-ps support for the effects defined in the `rotate` extension.

Answers to all exercises

Answer to exercise 1 (p.5): In the default setup they are all denote the reference point of the Xy-picture but the cartesian coordinate ⟨pos⟩ `(0,0)` denotes the point *origo* that may be changed to something else using the `:` operator.

Answer to exercise 2 (p.7): Use the ⟨pos⟩ition `<`X`,`Y`>+"ob"`.

Answer to exercise 3 (p.7): It first sets c according to "...". Then it changes c to the point right of c at the same distance from the right edge of c as its width, w, *i.e.*,

$$\underbrace{\text{The ...}}_{w}\underbrace{\qquad}_{w}\times$$

Answer to exercise 4 (p.8): The ⟨coord⟩ "`{"A";"B": "C";"D", x}`" returns the cross point. Here is how the author typeset the diagram in the exercise:

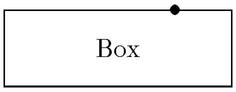
```
\xy
 %
 % set up and mark A, B, C, and D:
 (0,0)="A"  *\cir<1pt>{}*+!DR{A},
 (7,10)="B" *\cir<1pt>{}*+!DR{B},
 (13,8)="C" *\cir<1pt>{}*+!DL{C},
```

```
 (15,4)="D" *\cir<1pt>{}*+!DL{D},
 %
 % goto intersection and name+circle it:
 {"A";"B":"C";"D",x} ="I" *\cir<3pt>{},
 %
 % make dotted lines:
 "I";"A"**{} +/1pc/;-/1pc/ **\dir{..},
 "I";"D"**{} +/1pc/;-/1pc/ **\dir{..}
 %
\endxy
```

Answer to exercise 5 (p.8): To copy the p value to c, *i.e.*, equivalent to "p".

Answer to exercise 6 (p.8): When using the kernel connections that are all straight there is no difference, *e.g.*, `**{}?<` and `**{}+E` denote exactly the same position. However, for other connections it is not necessarily the case that the point where the connection enters the current object, denoted by `?<`, and the point where the straight line from p enters the object, denoted by `+E`, coincide.

Answer to exercise 7 (p.8): The code typesets the picture



Answer to exercise 8 (p.8): `s0` contains D and `s1` contains A.

Answer to exercise 9 (p.9): This does the job, saving each point to make the previous point available for the next piece:

```
\xy
 @i @+(0,-10) @+(10,3) @+(20,-5),
 s0="prev" @@{;"prev";**\dir{-}="prev"}
\endxy
```

Notice how we first save `s0` because that will be the last point that we run through thus the line is closed.

Answer to exercise 10 (p.9): The author used

```
\xy ={.{+DL(2)}.{+UR(2)}}"dbl",
 *+<3pc,2pc>{+}*\frm{.}, "dbl"*\frm{--}
\endxy
```

to typeset the figure in the exercise.

Answer to exercise 11 (p.10): The first typesets "a" centered around `0` and then moves c to the lower right corner, the second typesets "a" above the `0` point and does not change c. With a "+" at `0` they look like this:  and .

Answer to exercise 12 (p.10): They have the outlines

and

because the first is enlarged by the positive offset to the upper right corner and the second by the negative offset to the lower left corner.

Answer to exercise 13 (p.12): The first has no effect since the direction is set to be that of a vector in the current direction, however, the second reverses the current direction.

Answer to exercise 14 (p.15): One way is

```
$$\xy
 *{+}; p+(6,3)*{+} **{} ?(1)
 *\dir{-}  *!/-5pt/^\dir{-}
 *^\dir{-} *!/^-5pt/\dir{-}
\endxy$$
```

Thus we first create the two +s as p and c and connect them with the dummy connection `**{}` to setup the direction parameters. Then we move 'on top of c' with `?(1)` and position the four sides of the square using `^` and `_` for local direction changes and /⟨dimen⟩/ for skewing the resulting object by moving its reference point in the opposite direction.

Answer to exercise 15 (p.15): One way is to add extra half circles skewed such that they create the illusion of a shade:

```
$$\xy
 *\cir<5pt>{}
 *!<-.2pt,.2pt>\cir<5pt>{dr^ul}
 *!<-.4pt,.4pt>\cir<5pt>{dr^ul}
 *!<-.6pt,.6pt>\cir<5pt>{dr^ul}
\endxy$$
```

Answer to exercise 16 (p.17): This is the code that was actually used:

```
\xy (0,20)*[o]+{A};(60,0)*[o]+{B}="B"
**\crv{} \POS?(.4)*_+!UR{0},"B"
**\crv{(30,30)} \POS?*^+!D{1},"B"
**\crv{(20,40)&(40,40)} \POS?*^+!D{2},"B"
**\crv{(10,20)&(30,20)&(50,-20)&(60,-10)}
\POS?*+^!UR{4} \endxy
```

Answer to exercise 17 (p.17): This is the code that was used to typeset the picture:

```
\xy (0,20)*+{A};(60,0)*+{B}
**\crv{(10,20)&(30,20)&(50,-20)&(60,-10)}
 ?<*\dir{<} ?>*\dir{>}
 ?(.65)*{\oplus} *!LD!/^-5pt/{x}
 ?(.65)/12pt/*{\oplus} *!LD!/^-5pt/{x'}
```

```
 ?(.28)*=0{\otimes}-/40pt/*+{Q}="q"
 +/100pt/*+{P};"q" **\dir{-}
\endxy
```

Answer to exercise 18 (p.17): Here is the code that was used to typeset the picture:

```
\def\ssz#1{\hbox{$_{^{#1}}$}}
\xy (0,0)*+{A};(30,-10)*+{B}="B",**\dir{-},
"B"**\crv{(5,20)&(20,25)&(35,20)}
 ?<(0)*\dir{<}="a" ?>(1)*\dir{>}="h"
 ?(.1)*\dir{<}="b" ?(.9)*\dir{>}="i"
 ?(.2)*\dir{<}="c" ?(.8)*\dir{>}="j"
 ?(.3)*\dir{<}="d" ?(.7)*\dir{>}="k"
 ?(.4)*\dir{<}="e" ?(.6)*\dir{>}="l"
 ?(.5)*\dir{|}="f",
 "a"*!RC\txt{\ssz{(\lt)}};
  "h"*!LC\txt{\ssz{\;(\gt)}},**\dir{.},
 "b"*!RD{\ssz{.1}};
  "i"*!L{\ssz{\;.9}},**\dir{-},
 "c"*!RD{\ssz{.2}};
  "j"*!L{\ssz{\;.8}},**\dir{-},
 "d"*!RD{\ssz{.3}};
  "k"*!L{\ssz{\;.7}},**\dir{-},
 "e"*!RD{\ssz{.4}};
  "l"*!LD{\ssz{.6}},**\dir{-},
 "f"*!D!/^-3pt/{\ssz{.5}}
\endxy
```

Answer to exercise 19 (p.19): Here is how:

```
\xy
 (0,0)  *++={A} *\frm{o} ;
 (10,7) *++={B} *\frm{o} **\frm{.}
\endxy
```

Answer to exercise 20 (p.19): The `*\cir {}` operation changes c to be round whereas `*\frm {o}` does not change c at all.

Answer to exercise 21 (p.19): Here is how:

```
\xy
 (0,0)  *+++{A} ;
 (10,7) *+++{B} **\frm{.}
 **\frm{^\}} ; **\frm{_\}}
\endxy
```

The trick in the last line is to ensure that the reference point of the merged object to be braced is the right one in each case.

Answer to exercise 22 (p.23): This is how the author specified the diagram:

```
\UseCrayolaColors
$$\xy\drop[*1.25]\xybox{\POS
(0,0)*{A};(100,40)*{B}**{}
 ?<<*[@_][red][o]=<5pt>{\heartsuit};
 ?>>>*[@_][Plum][o]=<3pt>{\clubsuit}
 **[|*][|.5pt][thicker]\dir{-},
?(.1)*[left]!RD\txt{label 1}*[red]\frm{.}
?(.2)*[!gsave newpath
  xyXpos xyYpos moveto 50 dup rlineto
  20 setlinewidth 0 0 1 setrgbcolor stroke
  grestore][psxy]{.},
?(.2)*[@]\txt{label 2}*[red]\frm{.},
?(.2)*[BurntOrange]{*},
?(.3)*[halfsize]\txt{label 3}*[red]\frm{.}
?(.375)*[flip]\txt{label 4}*[red]\frm{.}
?(.5)*[dblsize]\txt{label 5}*[red]\frm{.}
?(.5)*[WildStrawberry]{*},
?(.7)*[hflip]\txt{label 6}*[red]\frm{.}
?(.8)*[vflip]\txt{label 7}*[red]\frm{.}
?(.9)*[right]!LD\txt{label 8}*[red]\frm{.}
?(.5)*[@][*.66667]!/^30pt/
 \txt{special effect: aligned text}
 *[red]\frm{.}
}\endxy$$
```

Answer to exercise 23 (p.27): Here is what the author did:

```
\xy *+{A}*\cir<10pt>{}="me"
 \PATH ~={**{}} ~-{**dir{-}}
  `ul^ur,"me" "me" |>*:(1,-.15)\dir{>}
\endxy
```

The trick is getting the arrow head right: the : modifier to the explicit `\dir` ⟨object⟩ does that.

Answer to exercise 24 (p.27): The author did

```
\xy(0,0)
 \ar @{-->} (30,7) ^A="a"
 \POS(10,12)*+\txt{label} \ar "a"
\endxy
```

Answer to exercise 25 (p.28): Here is the entire XY-picture of the exercise:

```
\xy ;<1pc,0pc>:
 \POS(0,0)*+{A}
 \ar    +(-2,3)*+{A'}*\cir{}
 \ar @2 +( 0,3)*+{A''}*\cir{}
 \ar @3 +( 2,3)*+{A'''}*\cir{}
 \POS(6,0)*+{B}
 \ar @1{||.>>} +(-2,3)*+{B'}*\cir{}
 \ar @2{||.>>} +( 0,3)*+{B''}*\cir{}
 \ar @3{||.>>} +( 2,3)*+{B'''}*\cir{}
\endxy
```

The first batch use the default `{->}` specification.

Answer to exercise 26 (p.28): The author used

```
\newdir{ >}{{}*!/-5pt/\dir{>}}
```

Answer to exercise 27 (p.29): The author used

```
\xy
 \ar @{>>*\composite{\dir{x}*\dir{+}}<<}
 (20,7)
\endxy
```

Answer to exercise 28 (p.29): The author used

```
\xy*{\bullet}="b"
 \ar
  @'{@+(10,10) @+(0,20) @+(-10,10)}
  "b"
\endxy
```

Answer to exercise 29 (p.31): Here is the code used to typeset the *pasting diagram* in figure 14.

```
\xymatrixrowsep{1.5pc}
\xymatrixcolsep{3pc}
\diagram
 &&\relax\rtwocell<0>^{f_3^{}\;\;}{\omit}
 &\relax\ddtwocell<0>{\omit}
  \drtwocell<0>^{\;\;f_4^{}}{<3>}
  \ddrrtwocell<\omit>{<8>}\\
&&&&\relax\drtwocell<0>^{\;\;f_5^{}}{\omit}\\
A \uurrlowertwocell<-6>{\omit}\relax
\uurrcompositemap<2>_{f_1^{}}^{f_2^{}}{<.5>}
 \drtwocell<0>_{g_1^{}\;}{\omit}
 &&&\relax\urtwocell<0>{\omit}
 &&\relax\rtwocell<0>^{f_6^{}\;}{\omit}
 &\relax\rlowertwocell<-3>_{g_4^{}}{<-1>}
 \rcompositemap<6>_{f_7^{}}^{f_8^{}}{\omit}
 & B \\
 &\relax\urrtwocell<0>{\omit}
 \xcompositemap[-1,4]{}%
  <-4.5>_{g_2^{}}^{g_3^{}}{\omit}\\
\enddiagram
```

For the straight arrows, it would have been simpler to use `\..to` provided `xyarrow` has been loaded. Instead `\..twocell<0>...{\omit }` was used to illustrate the versatility of nudging and `\omit` ; thus `xy2cell` can completely handle a wide range of diagrams, without requiring `xyarrow`. Note also the use of `\relax` at the start of each new cell, to avoid premature expansion of a complicated macro, which can upset the compiling mechanism.

Answer to exercise 30 (p.34): Here is the code used by the author to set the first diagram.

```
{\uppercurveobject{{?}}
 \lowercurveobject{{\circ}}
\xymatrixcolsep{5pc}
\xymatrixrowsep{2pc}
\diagram
 \relax\txt{ FUn }\rtwocell<8>{!\&}
 & \relax\txt{ gaMES }
 \enddiagram}
```

Here is the code used for the second diagram.

```
\xymatrixcolsep{2.5pc}
\xymatrixrowsep{4pc}
\diagram
 \relax\txt<1.5cm>{\bf Ground State}
 \rrtwocell<12>~^{+{}~**!/-2.5pt/\dir{>}}
 ~_{++{}~**!/5pt/\dir{<<}}
 ^{<1.5>\txt{\small continuous power}}
 _{<1.5>\txt{\small pulsed emission}}{!}
& \relax\;\; N\!i\,C\!d\;\; \Circled
& \relax\txt<1.50cm>{\bf Excited State}
\enddiagram
```

Answer to exercise 31 (p.36): A modifier was used to make all entries round and all entries had an extra circle added (these things are independent). Finally the matrix was rotated to make it possible to enter it as a simple square:

```
\entrymodifiers={[o]=<1pc>}
\everyentry={\drop\cir{}}
\xy\xymatrix@ur{
 A \save[];[r] **\dir{-},
[];[dr]**\dir{-},
[];[d] **\dir{-}\restore
   & B \\
 C & D }\endxy
```

Answer to exercise 32 (p.36): The author did

```
\xy\xymatrix{
 *+!/r1em/{A\times B}
     \ar[r]^{/A} \ar[d]_{/B}
 & B \ar[d]^{\times A}
\\
 A   \ar[r]_{B\times}
 & *+!/l1em/{B\times A}
}\endxy
```

Notice the use of a `+` modifier to ensure that the entries are grown just as in the default case.

Answer to exercise 33 (p.36): Here is how:

```
\objectheight{1pc} \objectwidth{3pc}
\xymatrixrowsep={0pc}
\everyentry={\framed}
\xy\xymatrix{%
 : \save+<-4pc,1pc>*\hbox{\it root}
     \ar[]
   \restore
\\
```

```
 {\bullet}
   \save*{}
    \ar`r[dd]+/r4pc/`[dd][dd]
   \restore
\\
 {\bullet}
   \save*{}
    \ar`r[d]+/r3pc/`[d]+/d2pc/
  `[uu]+/l3pc/`[uu][uu]
   \restore
\\
 1 }\endxy
```

Answer to exercise 34 (p.37): The first *A* was named to allow reference from the last:

```
\xygraph{
 []A="A1" :@/^.5pc/ [r]A
  :@/^.5pc/ [r]A
  :@/^1pc/ "A1" }
```

Answer to exercise 35 (p.39): Here is the code actually used to typeset the `\xypolygon` s, within an `\xygraph` . It illustrates three different ways to place the numbers. Other ways are also possible.

```
\def\objectstyle{\scriptscriptstyle}
\xy \xygraph{!{/r2pc/:}
[] !P3"A"{\bullet}
"A1"!{+U*++!D{1}} "A2"!{+LD*+!RU{2}}
"A3"!{+RD*+!LU{3}} "A0"
[rrr]*{0}*\cir<5pt>{}
!P6"B"{~<-\cir<5pt>{}}
"B1"1 "B2"2 "B3"3 "B4"4 "B5"5 "B6"6 "B0"
[rrr]0 !P9"C"{~*{\xypolynode}}}\endxy
```

References

[1] American Mathematical Society. *AMS-LaTeX Version 1.1 User's Guide*, version 1.1 edition, 1991. Available for anonymous from CTAN in `macros/ams/amslatex/doc`.

[2] Karl Berry. *Expanded plain TeX*, version 2.6 edition, May 1994. Available for anonymous from CTAN in `macros/eplain/doc`.

[3] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, 1994.

[4] Brian W. Kernighan. PIC—a language for typesetting graphics. *Software Practice and Experience*, 12(1):1–21, 1982.

[5] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1984.

[6] Donald E. Knuth. *Computer Modern Typefaces*, volume A of *Computers & Typesetting*. Addison-Wesley, 1986.

[7] Leslie Lamport. *LaTeX—A Document Preparation System*. Addison-Wesley, 1986.

[8] Leslie Lamport. *LaTeX—A Document Preparation System*. Addison-Wesley, 2nd edition, 1994.

[9] P. Naur et al. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3:299–314, 1960.

[10] Tomas Rokicki. *DVIPS: A TeX Driver*. Distributed with the dvips program found on CTAN archives.

[11] Kristoffer H. Rose. XY-pic complete sources with TeXnical commentary. To appear.

[12] Kristoffer H. Rose. How to typeset pretty diagram arrows with TeX—design decisions used in XY-pic. In Jiří Zlatuška, editor, *EuroTeX '92—Proceedings of the 7th European TeX Conference*, pages 183–190, Prague, Czechoslovakia, September 1992. Czechoslovak TeX Users Group.

[13] Kristoffer H. Rose. Typesetting diagrams with XY-pic: User's manual. In Jiří Zlatuška, editor, *EuroTeX '92—Proceedings of the 7th European TeX Conference*, pages 273–292, Prague, Czechoslovakia, September 1992. Czechoslovak TeX Users Group.

[14] Kristoffer H. Rose. XY-pic user's guide. Mathematics Report 94–148, MPCE, Macquarie University, NSW 2109, Australia, June 1994. For version 2.10+. Latest version available by anonymous ftp in `ftp.diku.dk: /diku/users/kris/TeX/xyguide.ps.Z`.

[15] Michael D. Spivak. *The Joy of TeX—A Gourmet Guide to Typesetting with the AMS-TeX Macro Package*. American Mathematical Society, second edition, 1990.