XY-pic Reference Manual

Kristoffer H. Rose ⟨kris@diku.dk⟩[×]

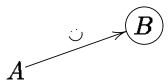
Ross Moore ⟨ross@mpce.mq.edu.au⟩[†]

Version 3.0+[‡] ⟨1995/07/20⟩

Abstract

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

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

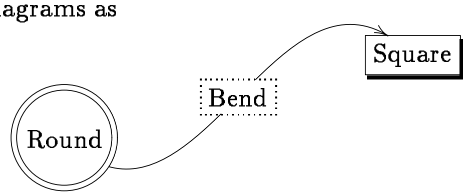


was drawn by the XY-pic kernel code

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

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

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

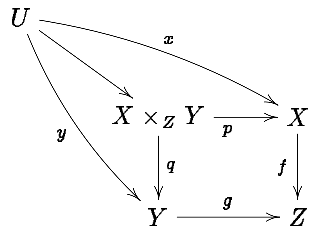


which was typeset by

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

using the 'curve' and 'frame' extensions.

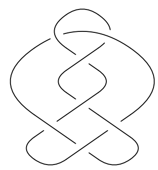
All this is made accesible through *features* that provide convenient notation such that users can enter special classes of diagrams in an intuitive form, *e.g.*, the diagram



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

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

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



The current implementation is programmed completely within "standard TEX and METAFONT", *i.e.*, using TEX macros (no `\specials`) and fonts designed using METAFONT. Optionally a special 'driver' 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.

[‡]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 registered Trademark of Adobe, Inc. [1].

Contents

Appendices

List of Figures

Please direct any question or suggestion for improvement directly to the author of the component in question, preferably by electronic mail using the indicated address. Complete documents and printed technical documentation or software is most useful. Please note note that XY-pic is free software (see the User's Guide or the accompanying file COPYING for details).

Kristoffer Rose

Ross Moore

Part I
The Kernel

Vers. 3.0+ by Kristoffer H. Rose ⟨kris@diku.dk⟩

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

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

Notation

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 Loading XY-pic

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

```
\input xy
```

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

The rest of this section describes things you must consider if you need to use XY-pic together with other

[2]BNF is the notation for "meta-linguistic formulae" first used in [10] to describe the syntax of the Algol programming language. We use it with the conventions of the TEXbook [6]: '⟶' 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'.

macro packages, style options, or formats. The less your environment deviates from plain TEX the easier it should be. Consult the TEXnical documentation [15] for the exact requirements for other definitions to coexist with XY-pic.

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

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

In general the XY-pic kernel will check all control sequences it redefines *except* that (1) generic temporaries like `\next` are not checked, (2) predefined font identifiers (see §1.3) are assumed intentionally preloaded, and (3) some of the more exotic control sequence names used internally (like `\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.* [6, p.37]): when loaded XY-pic requires the characters `␣\{}%` (the first is a space) to have their standard meaning and all other printable characters to have the *same category as when XY-pic will be used*—in particular this means that (1) you should surround the loading of XY-pic with `\makeatother` ... `\makeatletter` when loading it from within a LATEX package, and that (2) XY-pic should be loaded after files that change category codes (like the `german.sty` that makes `"` active).

However, it is possible to 'repair' the problem in case any of the characters `#$&'+-.<=>`` ` change category code:

`\xyresetcatcodes`

---

will load the file `xyrecat.tex` (version 3.0) to to do it.

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

xyidioms.tex: This included file provides some common idioms whose definition depends on the used format such that XY-pic can use predefined dimension registers etc. and yet still be independent of the format under which it is used. The current version (3.0) handles plain TEX (version 2 and 3 [6]), AMS-TEX (version 2.0 and 2.1 [16]), LATEX (version 2.09 [8] and 2ε [9]), AMS-LATEX (version 1.0, 1.1 [2], and 1.2), and eplain (version 2.6 [3])[3].

[3]The 'v2' feature introduces some name conflicts for compatibility.

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

`\usepackage [`⟨option⟩`,...] {xy}`

---

where the ⟨option⟩s will be interpreted as if passed to `\xyoption` (*cf.* §7); 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.* [4, table 11.2, p.317]) will invoke the appropriate backend extension.

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

1.2 Logo, version, and messages

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

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

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

1.3 Fonts

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

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

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

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

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

1.4 Allocations

One final thing that you must be aware of is 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 current kernel allocates 4 counters, 26 dimensions, 2 box registers, 2 token registers, 1 read channel, and 1 write channel (when running under LaTeX; some other formats use slightly more because the provided generic temporaries are used). Options may allocate further registers (currently loading *everything* loads 5 dimen-, 3 toks-, 1 box-, and 7 count-registers in addition to the kernel ones).

2 Picture basics

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

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

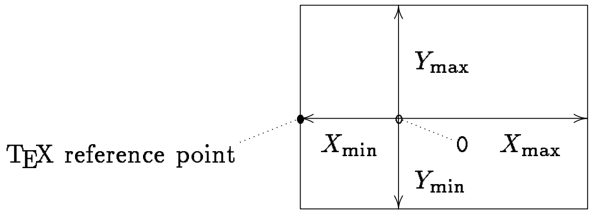
`\xy` ⟨pos⟩ ⟨decor⟩ `\endxy`

---

builds a box with an XY-picture (LaTeX users may substitute `\begin{xy}` ... `\end{xy}` if they prefer). ⟨pos⟩ and ⟨decor⟩ are components of the special 'graphic language' which XY-pictures are specified in. We explain the language components in general terms in this § and in more depth in the following §§.

2.1 Positions

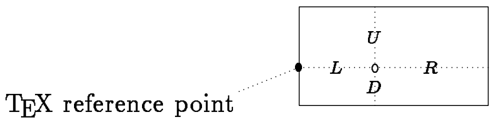
All *positions* may be written `<`X`,`Y`>` where X is the TeX dimension distance *right* and Y the distance *up* from the *zero position* `0` of the XY-picture (`0` has coordinates `<0mm,0mm>`, of course). The zero position of the XY-picture determines the box produced by the `\xy`...`\endxy` command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



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

2.2 Objects

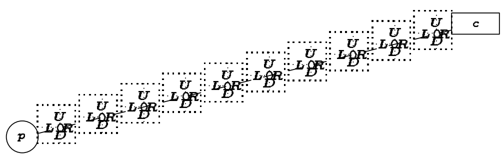
The simplest form of putting things into the picture is to 'drop' an *object* at a position. An object is like a TeX box except that it has a general *Edge* around its reference point—in particular this has the *extents* (*i.e.*, it is always contained within) the dimensions L, R, U, and D away from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



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

2.3 Connections

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



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

2.4 Decorations

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

2.5 The XY-pic state

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

Furthermore, XY-pic has a configurable *cartesian coordinate system* described by an *origin* position <X_{origin} ,Y_{origin}> and two *base vectors* <X_{xbase} ,Y_{xbase}> and <X_{ybase} ,Y_{ybase}> accessed by the usual notation using parentheses:

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

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

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

---

\everyxy={ ⟨text⟩ }

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

3 Positions

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

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

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

Notes

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

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

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

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

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

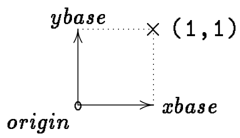
Note: non-rectangular objects are first "translated" into a rectangle by using a diagonal through the object as the diagonal of the rectangle.

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:

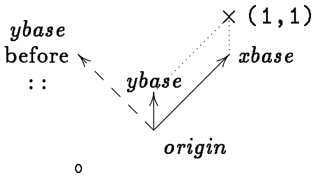
Syntax			Action
⟨pos⟩	$\longrightarrow$	⟨coord⟩	$c \leftarrow$ ⟨coord⟩
	|	⟨pos⟩ `+` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ + ⟨coord⟩[3a]
	|	⟨pos⟩ `-` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ − ⟨coord⟩[3a]
	|	⟨pos⟩ `!` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ then skew[3b] c by ⟨coord⟩
	|	⟨pos⟩ `.` ⟨coord⟩	$c \leftarrow$ ⟨pos⟩ but also 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⟩	$\longrightarrow$	⟨vector⟩	⟨pos⟩ is ⟨vector⟩ with zero size
	|	⟨empty⟩ | `c`	reuse last c (do nothing)
	|	`p`	p
	|	`x` | `y`	axis intersection[3k] with $\overline{pc}$
	|	`s`⟨digit⟩ | `s{`⟨number⟩`}`	stack[3o] position ⟨digit⟩ or ⟨number⟩ below the top
	|	`"`⟨id⟩`"`	restore what was saved[3p] as ⟨id⟩ earlier
	|	`{` ⟨pos⟩ ⟨decor⟩ `}`	the c resulting from interpreting the group[3l]
⟨vector⟩	$\longrightarrow$	`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⟩[3m]
⟨corner⟩	$\longrightarrow$	`L` | `R` | `D` | `U`	offset[3n] to left, right, down, up side
	|	`CL` | `CR` | `CD` | `CU` | `C`	offset[3n] to center of side, true center
	|	`LD` | `RD` | `LU` | `RU`	offset[3n] to actual left/down, ... corner
	|	`E` | `P`	offset[3n] to nearest/proportional edge point to p
⟨place⟩	$\longrightarrow$	`<` ⟨place⟩ | `>` ⟨place⟩	shave[3h] `(0)`/`(1)` to edge of p/c, $f \leftarrow$ `0`/`1`
	|	`(` ⟨factor⟩ `)` ⟨place⟩	$f \leftarrow$ ⟨factor⟩
	|	⟨slide⟩	pick place[3h] and apply ⟨slide⟩
	|	`! {`⟨pos⟩`}` ⟨slide⟩	intercept[3j] with line setup by ⟨pos⟩ and apply ⟨slide⟩
⟨slide⟩	$\longrightarrow$	`/` ⟨dimen⟩ `/`	slide[3i] ⟨dimen⟩ further along connection
	|	⟨empty⟩	no slide
⟨stacking⟩	$\longrightarrow$	`@` ⟨stack op⟩ ⟨coord⟩	do ⟨stack op⟩eration[3o] on ⟨coord⟩
	|	`@i` | `@(` | `@)`	init, enter, leave stack[3o]
⟨saving⟩	$\longrightarrow$	`= "`⟨id⟩`"`	save[3p] c as `"`⟨id⟩`"`
	|	`=`⟨macro⟩ `"`⟨id⟩`"`	define ⟨macro⟩[3q] `"`⟨id⟩`"`

Figure 1: ⟨pos⟩itions.

firstly `0;<1cm,0cm>:` sets the coordinate system



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



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

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

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

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

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

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

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

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

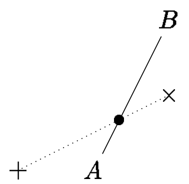
All this is probably best illustrated with some examples: each ⊗ in figure 2 is typeset by a sequence of the form p`;` c `**\dir{.}` `?`⟨place⟩ `*{\oplus}` where we indicate the `?`⟨place⟩ in each case. (We also give examples of ⟨slide⟩s.)

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

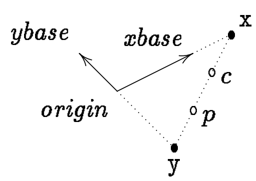
3j. This special ⟨place⟩ finds the point where the last connection intercepts with the line from p to c as setup by the ⟨pos⟩, thus usually this will have the form `!{`⟨coord⟩`;`⟨coord⟩`}`[5], for example,

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

will typeset



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



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

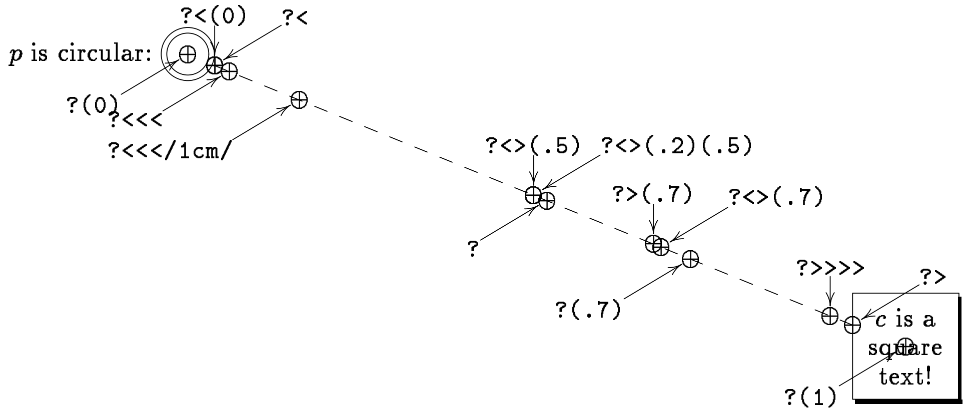
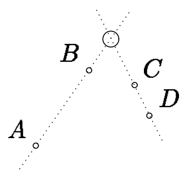


Figure 2: Example ⟨place⟩s

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



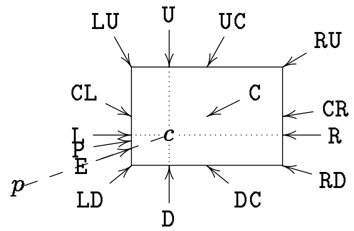
3l. A ⟨pos⟩ ⟨decor⟩ *grouped* in {}-braces[6] is interpreted in a local scope in the sense that any p and *base* built within it are forgotten afterwards, leaving only the c as the result of the ⟨coord⟩. **Note:** Only p and *base* are restored – it is not a TeX group.

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

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

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

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



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

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

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

Exercise 7: What does this typeset?

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

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

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

3o. The *stack* is a special construction useful for storing a sequence of ⟨pos⟩itions that are accessible using the special ⟨coord⟩inates `s`n, where n is either a single digit or a positive integer in `{}`s: `s0` is always the 'top' element of the stack and if the

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

stack has depth d then the 'bottom' element of the stack has number `s{`$d-1$`}`. The stack is said to be 'empty' when the depth is 0 and then it is an error to access any of the `s`n or 'pop' which means remove the top element, shifting what is in `s1` to `s0`, `s2` to `s1`, etc. Similarly, 'push c' means to shift `s0` to `s1`, etc., and then insert the c as the new `s0`.

The following syntax is used for such operations; all take a ⟨coord⟩ as argument and do something with it:

`@`⟨stack op⟩	use of ⟨coord⟩
`@+`	push ⟨coord⟩
`@-`	$c \leftarrow$ ⟨coord⟩ then pop
`@=`	load stack with ⟨coord⟩
`@@`	do ⟨coord⟩ for $c \leftarrow$ all stack elements

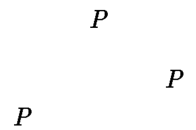
To 'load stack', means to load the entire stack with the positions set by ⟨coord⟩ within which `,` means 'push c'.

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

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

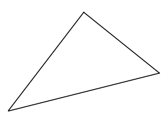
```
\xy
 @={(0,-10),(10,3),(20,-5)} @@{*{P}}
\endxy
```

will typeset



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

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



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

Using a `"`⟨id⟩`"` that was never defined is an error, however, saving into a name that was previously defined just replaces the definition, *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.

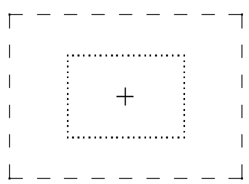
3q. The general form, `=`⟨macro⟩`"`⟨id⟩`"` can be used to save various things:

⟨macro⟩	effect when used
`:`	`"`⟨id⟩`"` restores current *base*
⟨coord⟩	`"`⟨id⟩`"` reinterprets ⟨coord⟩
`@`	`@="`⟨id⟩`"` reloads current stack

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

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

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



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

4 Objects

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

Notes

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

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

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

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

4g. A ⟨size⟩ is a pair $<W,H>$ of the width and height of a rectangle. When given as a ⟨vector⟩ these are just the vector coordinates, *i.e.*, the ⟨vector⟩ starts in the lower left corner and ends in the upper right corner. The 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 11: How are the objects typeset by the ⟨pos⟩itions "`*+UR{\sum}`" and "`*+DL{\sum}`" enlarged?

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

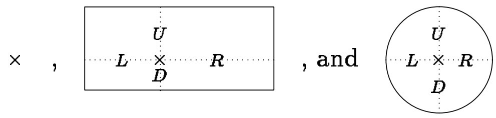
4h. A *hidden* object will be typeset but hidden from XY-pic in that it won't affect the size of the entire picture as discussed in §2.1.

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

<table>
<tr><th>Syntax</th><th></th><th></th><th>Action</th></tr>
<tr><td>⟨object⟩</td><td>⟶</td><td>⟨modifier⟩ ⟨object⟩</td><td>apply ⟨modifier⟩ to ⟨object⟩</td></tr>
<tr><td></td><td>|</td><td>⟨objectbox⟩</td><td>build ⟨objectbox⟩ then apply its ⟨modifier⟩s</td></tr>
<tr><td>⟨objectbox⟩</td><td>⟶</td><td><code>{</code> ⟨text⟩ <code>}</code></td><td>build default[4a] object</td></tr>
<tr><td></td><td>|</td><td>⟨library object⟩ | <code>@</code>⟨dir⟩</td><td>use ⟨library object⟩ or ⟨dir⟩ectional (see §6)</td></tr>
<tr><td></td><td>|</td><td>⟨TEX box⟩ <code>{</code> ⟨text⟩ <code>}</code></td><td>build box[4b] object with ⟨text⟩ using the given ⟨TEX box⟩ command, e.g., <code>\hbox</code></td></tr>
<tr><td></td><td>|</td><td><code>\object</code> ⟨object⟩</td><td>wrap up the ⟨object⟩ as a finished object box[4c]</td></tr>
<tr><td></td><td>|</td><td><code>\composite {</code> ⟨composite⟩ <code>}</code></td><td>build composite object box[4d]</td></tr>
<tr><td></td><td>|</td><td><code>\xybox {</code> ⟨pos⟩ ⟨decor⟩ <code>}</code></td><td>package entire XY-picture as object[4e]</td></tr>
<tr><td>⟨modifier⟩</td><td>⟶</td><td><code>!</code> ⟨vector⟩</td><td>⟨object⟩ has its reference point shifted[4f] by ⟨vector⟩</td></tr>
<tr><td></td><td>|</td><td><code>!</code></td><td>⟨object⟩ has the original reference point reinstated</td></tr>
<tr><td></td><td>|</td><td>⟨add op⟩ ⟨size⟩</td><td>change ⟨object⟩ size[4g]</td></tr>
<tr><td></td><td>|</td><td><code>h</code> | <code>i</code></td><td>⟨object⟩ is hidden[4h], invisible[4i]</td></tr>
<tr><td></td><td>|</td><td><code>[</code> ⟨shape⟩ <code>]</code></td><td>⟨object⟩ is given the specified ⟨shape⟩[4j]</td></tr>
<tr><td></td><td>|</td><td>⟨direction⟩</td><td>set current direction for this ⟨object⟩</td></tr>
<tr><td>⟨add op⟩</td><td>⟶</td><td><code>+</code> | <code>-</code> | <code>=</code> | <code>+=</code> | <code>-=</code></td><td>grow, shrink, set, grow to, shrink to</td></tr>
<tr><td>⟨size⟩</td><td>⟶</td><td>⟨empty⟩</td><td>default size[4g]</td></tr>
<tr><td></td><td>|</td><td>⟨vector⟩</td><td>size as sides of rectangle covering the ⟨vector⟩</td></tr>
<tr><td>⟨direction⟩</td><td>⟶</td><td>⟨diag⟩</td><td>⟨diag⟩onal direction[4k]</td></tr>
<tr><td></td><td>|</td><td><code>v</code> ⟨vector⟩</td><td>direction[4k] of ⟨vector⟩</td></tr>
<tr><td></td><td>|</td><td><code>{</code> ⟨pos⟩ ⟨decor⟩ <code>}</code></td><td>direction[4k] from p to c after ⟨pos⟩ ⟨decor⟩</td></tr>
<tr><td></td><td>|</td><td>⟨direction⟩ <code>:</code> ⟨vector⟩</td><td>vector relative to ⟨direction⟩[4l]</td></tr>
<tr><td></td><td>|</td><td>⟨direction⟩ <code>_</code> | ⟨direction⟩ <code>^</code></td><td>90° clockwise/anticlockwise to ⟨direction⟩[4l]</td></tr>
<tr><td>⟨diag⟩</td><td>⟶</td><td>⟨empty⟩</td><td>default diagonal[4k]</td></tr>
<tr><td></td><td>|</td><td><code>l</code> | <code>r</code> | <code>d</code> | <code>u</code></td><td>left, right, down, up diagonal[4k]</td></tr>
<tr><td></td><td>|</td><td><code>ld</code> | <code>rd</code> | <code>lu</code> | <code>ru</code></td><td>left/down, ... diagonal[4k]</td></tr>
<tr><td>⟨composite⟩</td><td>⟶</td><td>⟨object⟩</td><td>first object is required</td></tr>
<tr><td></td><td>|</td><td>⟨composite⟩ <code>*</code> ⟨object⟩</td><td>add ⟨object⟩ to composite object box[4d]</td></tr>
</table>

Figure 3: ⟨object⟩s.

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



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.

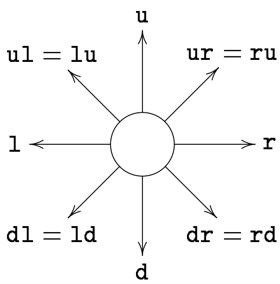
The default shape for objects is `[]` and for plain coordinates it is `[.]`.

Furthermore the ⟨shape⟩s `[r]`, `[l]`, `[u]`, and `[d]`, are defined for convenience to adjust the object to the indicated side by setting the reference point such that the reference point is the same distance from the opposite of the indicated edge and the two neighbour edges, *e.g.*, the object `[r]\hbox{Long text}` has reference point at the × in Long text. Finally, `[c]` puts the reference point at the center.

Note: Extensions may add new ⟨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`)`.

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

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

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

Adding a single `^` or `_` denotes the result of rotating the default direction a right angle in the positive and negative direction, *i.e.*, anti-/clockwise, respectively.

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

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

5 Decorations

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

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

Notes

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

```
\def\ToPOS{\save\afterPOS{%
  \POS**{}?>*\dir2{>}**\dir2{-}
 \restore};p,}
```

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

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!

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

Figure 4: ⟨decor⟩ations.

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 [15] for details about this!

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

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

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

5e. It is possible to save an intermediate form of commands that generate parts of an XY-picture to a file such that subsequent typesetting of those parts is significantly faster: this is called *compiling*.

There are two ways to do this. The easiest is to use `\xycompile` around the ⟨pos⟩ ⟨decor⟩ to be compiled. This will assign file names numbered consecutively with a ⟨prefix⟩ which is initially the expansion of `\jobname-` but may be set with

`\CompilePrefix{`⟨prefix⟩`}`

This has the disadvantage, however, that if additional compiled XY-pictures are inserted then all subsequent pictures will have to be recompiled. Using `\xycompileto{`⟨name⟩`}` gives an explicit name to the compiled file, namely ⟨name⟩`.xyc`. In both cases compiled files contain code to check that the compiled code still corresponds to the ⟨pos⟩ ⟨decor⟩ as well as efficient compiled code to redo it. If the ⟨pos⟩ ⟨decor⟩ has changed then the compilation is redone.

6 Kernel object library

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

6.1 Directionals

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

`\dir`⟨dir⟩

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

---

⟨dir⟩ ⟶ ⟨variant⟩{⟨main⟩}

---

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

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

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

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

Notes

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

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`

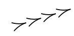
---

that will switch the use of such off and on.

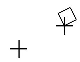
As can be seen by the last two columns, these (and most of the other connectors) also exist in double and triple versions with a 2 or a 3 prepended to the name. For convenience `\dir{=}` and `\dir{:}` are synonyms for `\dir2{-}` and `\dir2{.}`, respectively; similarly `\dir{==}` is a synonym for `\dir2{--}`.

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 Circle segments

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

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

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

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`	—	"M"

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:

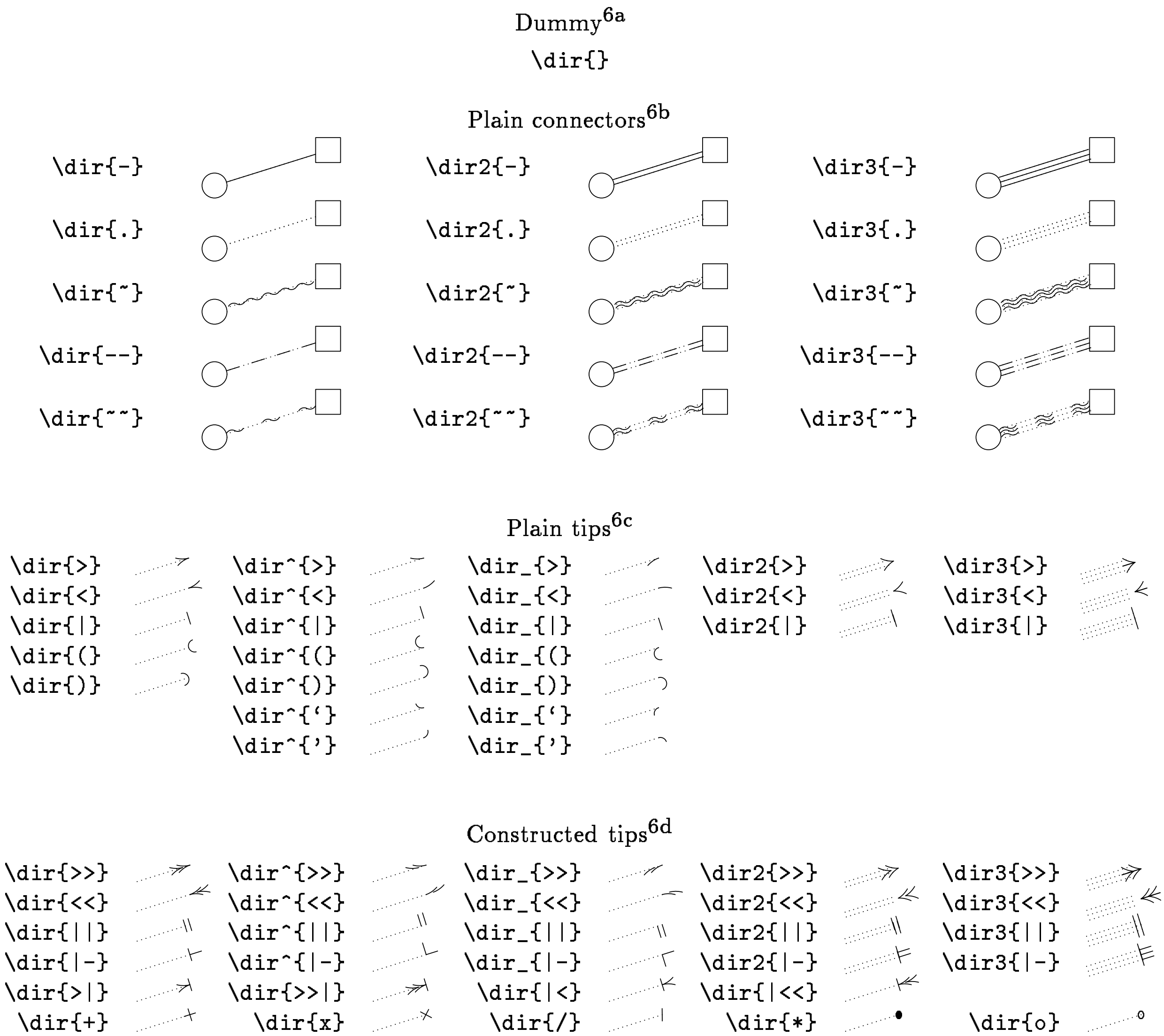


Figure 5: Kernel library ⟨dir⟩ectionals

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.

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

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

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

6.3 Text

Text in pictures is supported through the ⟨object⟩ construction

`\txt` ⟨width⟩ ⟨style⟩ `{`⟨text⟩`}`

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

7 XY-pic options

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

7.1 Interface

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

`\xyoption {` ⟨option⟩ `}`

`\xyrequire {` ⟨option⟩ `}`

`\xyoption` will cause the loading of 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. If the option file is not file then an attempt is made with ⟨option⟩ truncated to six characters.

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

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

which indicates that if the ⟨option⟩ is already loaded then ⟨code⟩ should be executed now, otherwise it should be saved and if ⟨option⟩ ever gets loaded then ⟨code⟩ should be executed afterwards. **Note:** The ⟨code⟩ should allow more than one execution.

7.2 Option file format

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

> `%%` ⟨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.

7.3 Backend options

Special care is taken to support backend options for particular drivers (*cf.* part IV). This is handled by the included file `xydriver.tex`.

xydriver.tex: This included file (version 3.0 by Ross Moore) allows for support for arbitrary dvi-driver applications.

The special thing about driver options is that initially loading a ⟨driver⟩-file simply declares the name of the requested driver and establishes what extensions this driver may support. Only if one of these extensions is also loaded, as an XY-pic option, will the TEX-environment be altered to support the extension using the requested ⟨driver⟩. The order in which the options are loaded is immaterial.

With some formats it is possible to delay this even further, until all requested ⟨driver⟩s and extensions are known. For example, using LATEX 2_ε all driver-specific changes are delayed until the `\begin{document}` line. This allows the ⟨driver⟩ to be changed (e.g. even if a format file has one pre-loaded).

With simpler formats, such as plain TEX, changes occur as soon as compatible ⟨driver⟩-extension pairs are found. If this results in an incorrect set of capabilities, then the command `\xyReloadDrivers` will force all the ⟨driver⟩-specific information to be re-setup immediately, provided this occurs later than the original setup. (This should only be necessary when using a format file which contains ⟨driver⟩-specific information that is not required, or perhaps when multiple drivers are needed.) The command `\xyShowDrivers` will cause a listing to be written to the Log-file, indicating what ⟨driver⟩-specific information will be set-up, accordinging to the currently requested options. This information has the form of `\xy`⟨driver⟩`@xy@`⟨extension⟩. When the same ⟨extension⟩ occurs multiple times, later occurring instances override the earlier ones.

`\xyoption{`⟨driver⟩`}`	loads a driver-file
`\xyoption{`⟨extension⟩`}`	loads extension
`\xyShowDrivers`	writes ⟨driver⟩-⟨extension⟩ pairs to the log-file
`\xyReloadDrivers`	resets driver information
`\MultipleDrivers`	allows multiple ⟨driver⟩s
`\UseSingleDriver`	forces one ⟨driver⟩ only

Default behaviour is to allow only a single ⟨driver⟩; loading a second ⟨driver⟩-file replaces the previous with the new one. The command, `\MultipleDrivers`, within the document preamble, changes this to allow multiple ⟨driver⟩s. Conversely `\UseSingleDriver` forces only the last-loaded ⟨driver⟩ to have any effect. With both of these commands it may also be necessary to `\xyReloadDrivers` after all options have been loaded, if there is no delaying mechanism as discussed above.

Note: The interface to the drivers is still under development and may change.

Part II
Extensions

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

Many of these are only fully supported when a suitable *driver* option (described in part IV) is also loaded.

8 Curve and Spline extension

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{curve}`

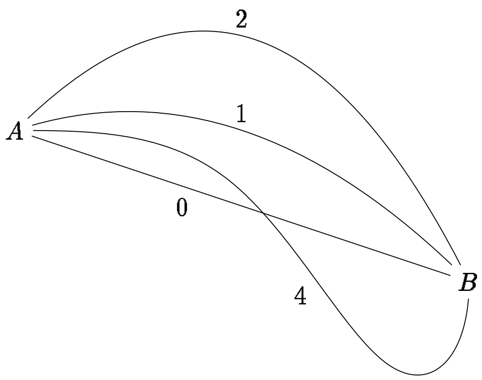
This option provides XY-pic with the ability to typeset spline curves by constructing curved connections using arbitrary directional objects and by encircling objects similarly. *Warning*: Using curves can be quite a strain on TEX's memory; you should therefore limit the length and number of curves used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

8.1 Curved connections

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

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

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



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

The previous picture was typeset using:

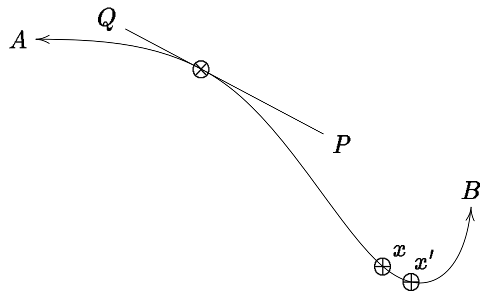
```
\xy (0,20)*+{A};(60,0)*+{B}
**\crv{}
**\crv{(30,30)}
**\crv{(20,40)&(40,40)}
**\crv{(10,20)&(30,20)&(50,-20)&(60,-10)}
\endxy
```

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

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

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

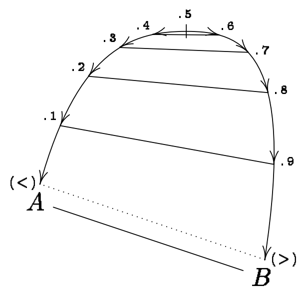
Using `?` will set the current direction to be tangential at that ⟨place⟩, and one can ⟨slide⟩ specified distances along the curve from a found ⟨place⟩ using the `?.../`⟨dimen⟩`/` notation:



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

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

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



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

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

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

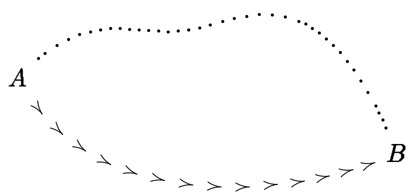
Syntax			Action
`\curve`⟨modifier⟩`{`⟨curve-object⟩⟨poslist⟩`}`			construct curved connection
⟨modifier⟩	⟶	⟨empty⟩	zero or more modifiers possible; default is `~C`
	|	`~`⟨curve-option⟩ ⟨modifier⟩	set ⟨curve-option⟩
⟨curve-option⟩	⟶	`p` | `P` | `l` | `L` | `c` | `C`	show only[8d] control points (`p`=points), 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 "connect" object[8b] and maybe more[8c]
⟨poslist⟩	⟶	⟨empty⟩ | ⟨pos⟩ ⟨delim⟩ ⟨poslist⟩	list of positions for control points
	|	`~@` | `~@` ⟨delim⟩ ⟨poslist⟩	add the current stack[8h] to the control points
⟨delim⟩	⟶	`&`	allowable delimiter

Figure 7: Syntax for curves.

Notes

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

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




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

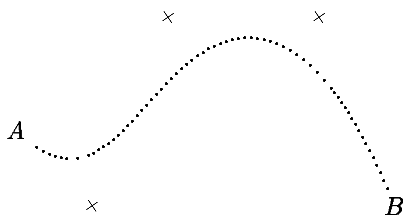
When there is no "connect" object then the tangent calculations are not carried out, resulting in a saving of time and memory; this is the default behaviour.

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

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

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

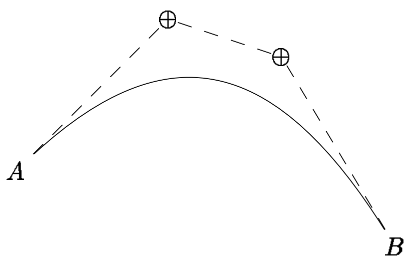
8f. By default the control points are marked with a small cross, specified by `*\dir{x}`. The "connect" object is ignored completely.



was typeset by ...

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

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



was typeset by ...

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

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

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

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

---

allow adjustment of the tolerance used to typeset curves. The first sets tolerance to .8pt, after which `\splinetolerance{0pt}` resets to the original default of fine curves.

8.2 Circles and Ellipses

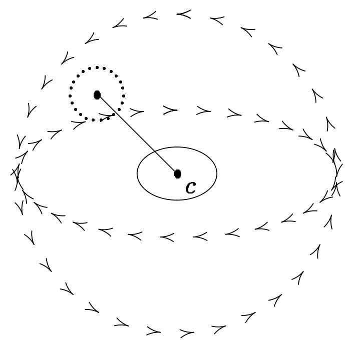
Here we describe the means to a specify circles of arbitrary radius, drawn with arbitrary line styles. When large-sized objects are used they are regularly spaced around the circle. Similarly ellipses may be specified, but only those having major/minor axes aligned in the standard directions; spacing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

Such a circle or ellipse is specified using...

`\xycircle`⟨vector⟩`{`⟨style⟩`}`

---

where the components of the ⟨vector⟩ determine the lengths of the axis for the ellipse; thus giving a circle when equal. The ⟨style⟩ can be any ⟨conn⟩, as in 15 that works with curved arrows—many do. Alternatively ⟨style⟩ can be any ⟨object⟩, which will be placed equally-spaced about the circle at a separation to snugly fit the ⟨object⟩s. If ⟨empty⟩ then a solid circle or ellipse is drawn.




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

9 Frame and Bracket extension

Vers. 3.0 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{` ⟨frame⟩ `}`

---

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

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

Framed with \frm{} frame[9a] | Framed with \frm{.} frame[9b] | Framed with \frm<44pt>{.} frame[9b]

These are overlayed with the \frm{.} frame above to show the way they are centered on the object

Framed with \frm{-} frame[9b] | Framed with \frm<8pt>{-} frame[9b] | Framed with \frm<44pt>{-} frame[9b]

Framed with \frm{=} frame[9b] | Framed with \frm<8pt>{=} frame[9b] | Framed with \frm<44pt>{=} frame[9b]

Framed with \frm{--} frame[9b] | Framed with \frm{o-} frame[9b] | Framed with \frm<44pt>{--} frame[9b]

Framed with \frm{,} frame[9c] | Framed with \frm<5pt>{,} frame[9c] | Framed with \frm{-,} frame[9c]

Framed with \frm{o} frame[9d] | Framed with \frm<8pt>{o} frame[9d] | Framed with \frm{.o} frame[9d]

Framed with \frm{oo} frame[9d] | Framed with \frm<8pt>{oo} frame[9d] | Framed with \frm{-o} frame[9d]

Framed with \frm{e} frame[9e] | Framed with \frm<20pt,8pt>{e} frame[9e] | Framed with \frm{.e} frame[9e]

Framed with \frm{ee} frame[9e] | Framed with \frm<20pt,8pt>{ee} frame[9e] | Framed with \frm{-e} frame[9e]

Figure 8: Plain ⟨frame⟩s.

Framed with \frm{_\}} frame[9f] | Framed with \frm{^\}} frame[9f] | Framed with \frm{\{} frame[9f] | Framed with \frm{\}} frame[9f]

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

Figure 9: Bracket ⟨frame⟩s.

9.1 Frames

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

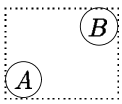
Notes

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

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

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

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

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

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

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

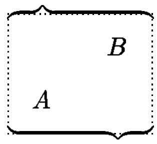
9.2 Brackets

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

Notes

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

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



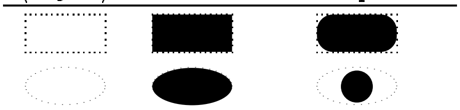
9g. *Parenthesis* are like braces except they have no nib and thus do not depend on where the reference point of *c* is.

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

9.3 Filled regions

In addition to the above there is a special frame that "fills" the inside of the current object with ink: \frm {*}. Some alteration to this shape is possible, using *\frm<dimen>{*}. Hence rectangular, oval, circular and elliptical shapes can be specified for filling. The following illustrates this in each case:

⟨object⟩	\frm{*}	\frm<6pt>{*}



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

9.4 Framing as object modifier

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

If shape modifiers should be applied to the ⟨frame⟩ alone then they can be included using `:` as separator: `[F-:red]` will make a red frame (provided the color extension is active, of course).

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

10 Computer Modern tip extension

Vers. 3.0 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 [7] and [6, appendix F]). These are often more pleasing in connection with curved arrows.

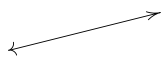
The user can switch the "computer modern" versions of the directionals shown in figure 10 on and off with these declarations:

```
\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. 3.0 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

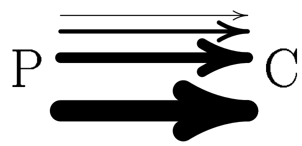
Using this extension will have no effect on the output unless used with a backend that explicitly supports it.

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.

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

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

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




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

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

The diagram in figure 11, page 28, uses different line-thicknesses and colours.

Poly-lines By a 'poly-line' we mean a path built from straight line segments having no gaps where each segment abuts the next. The poly-line could be the

Plain Computer Modern tips

`\dir{>}`	`\dir^{>}`	`\dir_{>}`
`\dir{<}`	`\dir^{<}`	`\dir_{<}`

Constructed Computer Modern tips

`\dir{>>}`	`\dir^{>>}`	`\dir_{>>}`	
`\dir{<<}`	`\dir^{<<}`	`\dir_{<<}`	
`\dir{>|}`	`\dir{>>|}`	`\dir{|<}`	`\dir{|<<}`

Figure 10: Computer Modern ⟨dir⟩ectionals

edges of a polygon, either closed or open if the end-points are different.

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

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

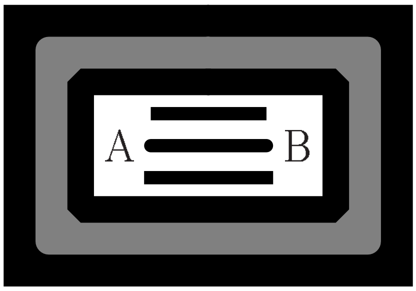
`[|J`⟨val⟩`]`	join style, ⟨val⟩ = 0, 1 or 2
`[mitre]`	mitre-join, same as `[|J0]`
`[roundjoin]`	round join, same as `[|J1]`
`[bevel]`	bevel-join, same as `[|J2]`
`[|C`⟨val⟩`]`	end-cap, ⟨val⟩ = 0, 1 or 2
`[butt]`	"butt" cap, same as `[|C0]`
`[roundcap]`	round cap, same as `[|C1]`
`[projcap]`	"projecting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

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

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

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

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




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

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

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

`\xypolyline{}`	solid line
`\xypolyline{.}`	dotted line

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

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

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

12 Rotate and Scale extension

Vers. 3.0 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⟩
[*⟨num⟩$_x$,⟨num⟩$_y$]	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; this depends on the coordinate system used by the driver, when 'raw' PostScript code is included.

13 Colour extension

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{color}`

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

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

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

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

`[`⟨colour name⟩`]`	use named colour
`\newxycolor{`⟨name⟩`}{`⟨code⟩`}`	define 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 $0 \le r, g, b, c, m, y, k, s \le 1$. The operators link to the built-in colour models or, in the case of `cmyk` for earlier versions of PostScript, give a simple emulation in terms of the RGB model.

Saving colour and styles When styles are saved using `[|=`⟨word⟩`]`, see §15, 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 `crayon` option, in which more colours are defined. Consult the file `xyps-col.doc` for the colours and their specifications in the RGB or CMYK models.

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

14 Pattern and Tile extension

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{tile}`

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

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

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

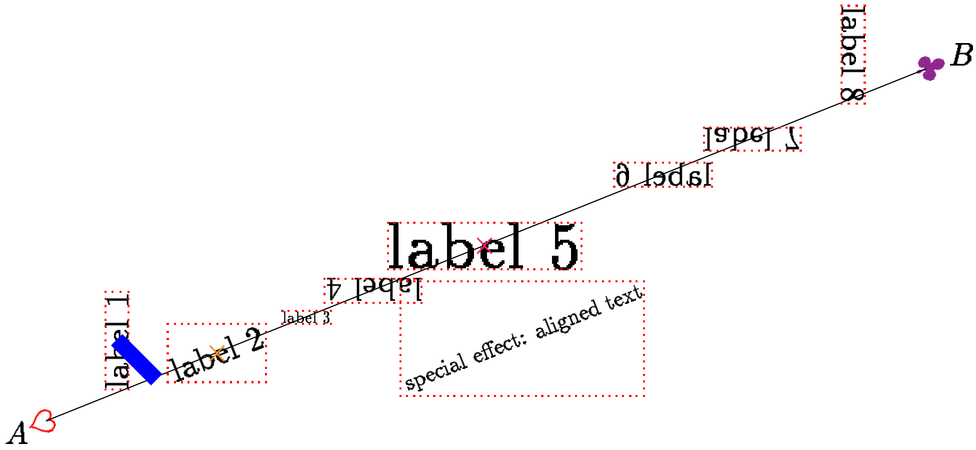


Figure 11: Rotations, scalings and flips

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

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

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

`[`⟨name⟩`]` use named pattern

`\newxypattern{`⟨name⟩`}{`⟨data⟩`}`
  specify new pattern using ⟨data⟩

`\UsePatternFile{`⟨file⟩`}`
  sets default file for patterns

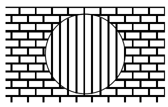
`\LoadAllPatterns{`⟨file⟩`}`
  load all patterns in ⟨file⟩

`\LoadPattern{`⟨name⟩`}{`⟨file⟩`}`
  load named pattern from ⟨file⟩

`\AliasPattern{`⟨alias⟩`}{`⟨name⟩`}{`⟨file⟩`}`
  let ⟨alias⟩ denote pattern from ⟨file⟩.

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

For example, the following picture



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

```
\AliasPattern{bricks}{mac12}{xymacpat}
\AliasPattern{bars}{mac08}{xymacpat}
\xy *+<5pc,3pc>{},{*[bricks]\frm{*}}
 ,*+<2.5pc>[o]{},*[bars]\frm{*},*\frm{o}
\endxy
```

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

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

The file `xymacpat.xyp` contains defining data for the 38 standard patterns available with the Macintosh Operating system. Figure 12 displays all these patterns.

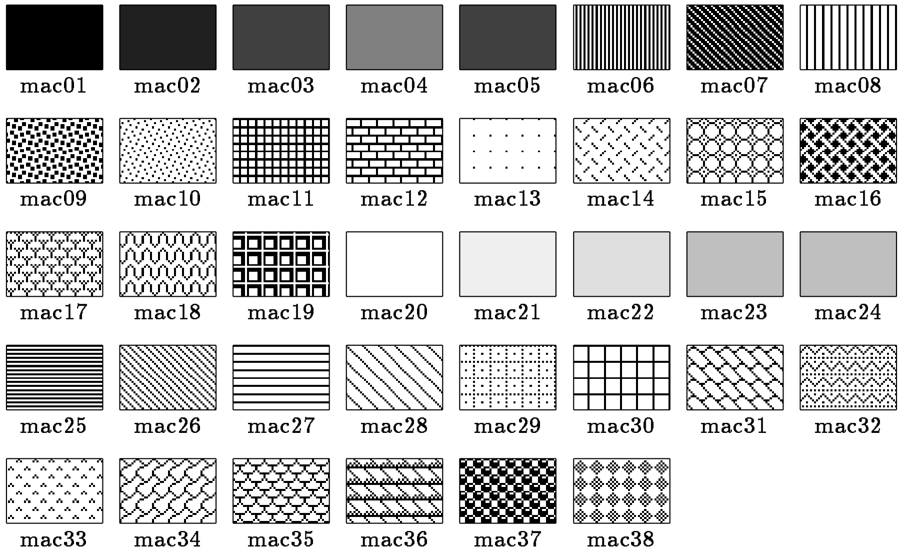


Figure 12: The 38 standard Macintosh patterns.

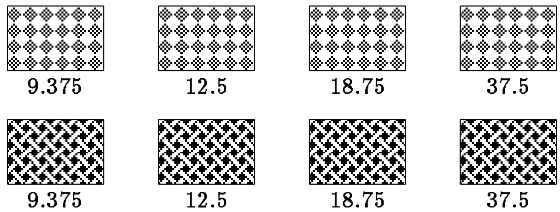
Rotating and Resizing Patterns Some implementations of patterns are sufficiently versatile to allow extra parameters to affect the way the pattern data is interpreted. PostScript is one such implementation in which it is possible to rotate the whole pattern and even to expand or contract the sizes of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

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

Special PostScript operators `pa` and `pf` set the pattern angle (normally 0) and 'frequency' measured in *cells per inch*. Hence, when used as an ⟨object⟩-modifier, `[! 30 pa 18.75 pq]` rotates the pattern by $30°$ clockwise and uses a smaller pattern cell (larger frequency). The default frequency of $12.5 = 300/(8\times3)$ means that each pixel in a pattern cell corresponds, on a device of resolution 300dpi, to a 3×3 square of device pixels; on such a device 18.75 uses 2×2 squares.

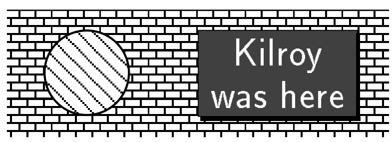
At 300dpi a frequency of $9.375 = 300/(8\times4)$ uses 4×4 squares. These match the natural size for pixels on a 75dpi screen and are pretty close for 72dpi screens. Though appropriate for screen displays, these are 'too chunky' for high quality printed work. Doubling the frequency is too fine for some patterns, hence the intermediate choice of 12.5 as default. In order for printed output to match the screen view, a PostScript operator `macfreq` has been defined to facilitate requests for 9.375, via `[!macfreq]`.

The next diagram displays changes to the frequency.



Saving patterns: When styles are saved using `[|=`⟨word⟩`]`, see §15, then the current pattern (if any) is also saved. Subsequent use of `[`⟨word⟩`]` recovers the pattern as well as colour and line-style settings. This includes any explicit variations applied using the "Style Escape" mechanism.

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




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

15 Style modifier extension

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{style}`

This extension provides the infrastructure used by other features which allow the user to request various effects related to the appearance of ⟨object⟩s; *e.g.*. thickness, rotation, non-standard dashing and colour.

Such effects 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. The purpose of this extension is to provide a framework whereby multiple effects requested for a single object may be collected together and applied without conflict, and within which new effects can be accommodated in a coherent way.

Saving styles: Once specified for an ⟨object⟩, the collection of styles can be assigned a name, using `[|=`⟨word⟩`]`. Then `[`⟨word⟩`]` becomes a new style, suitable for use with the same or other ⟨objects⟩s. Use a single ⟨word⟩ built from ordinary letters. 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.

16 PostScript backend

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{ps}`

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

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

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

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

16.1 Choosing the DVI-driver

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

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

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

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

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

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

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

Note: Previous versions of XY-pic loaded the POSTSCRIPT backend and driver simultaneously via a command of the form \UsePSspecials{⟨driver⟩}. For backward-compatibility this command still works, loading the latest version of the given ⟨driver⟩. However its future use is discouraged in favour of the option-loading mechanism, via xyoption{⟨driver⟩}. This latter mechanism is more flexible, both in handling upgrades of the actual driver support and in being extensible to support more general forms of \special commands.

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

\NoPSspecials	cancels POSTSCRIPT
\UsePSspecials {}	restores POSTSCRIPT

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

16.2 Why use POSTSCRIPT

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

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

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

- Circles and circle segments can be set for arbitrary radii.
- 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.[7]
- 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 following commands are available...

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

Normally it is sufficient to invoke \UsePSheader{}, which will use the default dictionary name of xy30+dict.pro, referring to the current version of XY-pic. The optional ⟨filename⟩ allows a different file to be used. Placing \dumpPSdict{..} within the document preamble causes the dictionary to be written to the supplied ⟨filename⟩.

[7]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.

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.

17 TPIC backend

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

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

Use of TPIC `\special`s offers an alternative to the XY-pic fonts. However they require a dvi-driver that is capable of recognizing and interpreting them. One such viewer is `xdvik`, Karl Berry's modification to the `xdvi` viewer on UNIX[8] systems running X-windows or a derivative. Also `dvipsk`, Karl Berry's modification to `dvips` also handles TPIC `\special`s, so `xdvik`/`dvipsk` is ideal for good quality screen-display and POSTSCRIPT printing.

18 Import graphics extension

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{import}`

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

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

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

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

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

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

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

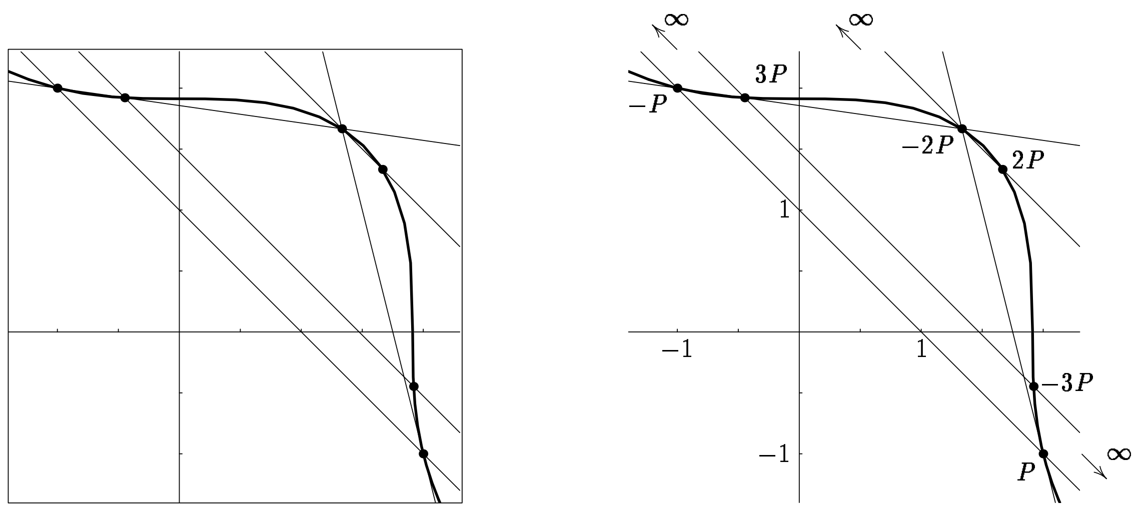
When provided, `(`*x-off*`,`*y*-off`)` give the distance in coordinate units from bottom-left corner to where the origin of coordinates should be located, usually within area covered by the ⟨graphic⟩. Usually the negatives of these numbers will give the coordinate location of the bottom-left corner of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

Normally the `\xyimport` command is used at the beginning of an `\xy..\endxy` environment. It is not necessary to give any basis setup, for this is deduced by measuring the dimensions of the ⟨graphic⟩ and using the supplied *width*, *height* and offsets. The ⟨graphic⟩ itself defines named ⟨pos⟩ called `"import"`, located at the origin and having appropriate extents to describe the area covered by the ⟨graphic⟩. This makes it particularly easy to surround the ⟨graphic⟩ with a frame, as on the left side of figure 13, or to draw axes passing through the origin.

Here is the code used to apply the labelling in figure 13:

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

[8] UNIX is a trademark of Bell Labs.



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

Figure 13: importing a graphic for labelling

```
,(0,1)*+!R{1},(0,-1)*+!R{-1}
,(2,-1)*+!RU{P},(-1,2)*+!RU{-P}
,(1.3333,1.6667)*+!UR{-2P}
,(1.6667,1.3333)*!DL{\;2P}
,(-.5,1.9)*++!DL{3P},(1.9,-.5)*!DL{\;-3P}
,(-1,2.3)*+++!D{\infty}*=0{},{\ar+(-.2,.2)}
,(.5,2.3)*+++!D{\infty}*=0{},{\ar+(-.2,.2)}
,(2.3,-1)*+++!L{\infty}*=0{},{\ar+(.2,-.2)}
\endxy
```

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

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

Part III

Features

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

The first two, 'all' and 'dummy', described in §§19 and 20, are trivial features that nevertheless prove useful sometimes. The next two, 'arrow' and '2cell', described in §21 and 22, provide special commands for objects that 'point'. The following, 'matrix' in §23, 'graph' in §24, 'poly' in §25, and 'knot' in §28, are *input modes* that support different overall structuring of (parts of) XY-pictures.

19 All features

Vers. 3.0 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{all}`

As a special convenience, this feature loads a subset of XY-pic[9], namely the following extensions: `curve` (*cf.* §8), `frame` (§9), `cmtip` (§10), `line` (§11), `rotate` (§12), `color` (§13), and the following features: `matrix` (§23), `arrow` (§21), and `graph` (§24).

20 Dummy option

Vers. 3.0 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{dummy}`

This option is provided as a template for new options, it provides neither features nor extensions.

21 Arrow and Path feature

Vers. 3.0 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: `\xyoption{arrow}`

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

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

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

§21.1 is about the `\PATH` primitive, including the syntax of paths, and §21.2 is about the `\ar` customisation of paths to draw arrows using XY-pic directional objects.

21.1 Paths

The fundamental commands of this feature are `\PATH` and `\afterPATH` that will parse the ⟨path⟩ according to the grammar in figure 14 with notes below.

Notes

21a. An ⟨action⟩ can be either of the characters `=<>-/`. The associated ⟨stuff⟩ is saved and used to call

`\PATHaction`⟨action⟩`{`⟨stuff⟩`}`

at specific times while parsing the ⟨path⟩:

⟨action⟩	applied...
=	before every segment
<	before next segment
>	before last segment
-	for every subsegment
/	after every segment

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 21d), 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 §21.2 below is little more than a special `\PATHaction` command and a clever defaulting mechanism.

21b. 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.

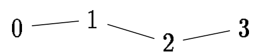
21c. Specifying `~{`⟨stuff⟩`}` will set the "failure continuation" to ⟨stuff⟩. This will be inserted when the last ⟨segment⟩ is expected—it can even replace it or add more ⟨segment⟩s, *i.e.*,

```
\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.

21d. 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 with the `~+`-defined ones) are interpreted and typeset as described below.

<table>
<tr><th colspan="3">Syntax</th><th>Action</th></tr>
<tr><td colspan="3"><code>\PATH</code> ⟨path⟩
<code>\afterPATH{</code>⟨decor⟩<code>}</code> ⟨path⟩</td><td>interpret ⟨path⟩
interpret ⟨path⟩ and then run ⟨decor⟩</td></tr>
<tr><td>⟨path⟩</td><td>⟶</td><td><code>~</code> ⟨action⟩ <code>{</code> ⟨stuff⟩ <code>}</code> ⟨path⟩</td><td>set ⟨action⟩[21a] to ⟨stuff⟩</td></tr>
<tr><td></td><td>|</td><td><code>~ + {</code> ⟨labels⟩ <code>}</code> ⟨path⟩</td><td>set default ⟨labels⟩[21b]</td></tr>
<tr><td></td><td>|</td><td><code>~ {</code> ⟨stuff⟩ <code>}</code> ⟨path⟩</td><td>set failure continuation[21c] to ⟨stuff⟩</td></tr>
<tr><td></td><td>|</td><td><code>'</code> ⟨segment⟩ ⟨path⟩</td><td>make straight segment[21d]</td></tr>
<tr><td></td><td>|</td><td><code>`</code> ⟨turn⟩ ⟨segment⟩ ⟨path⟩</td><td>make turning segment[21f]</td></tr>
<tr><td></td><td>|</td><td>⟨segment⟩</td><td>make last segment[21g]</td></tr>
<tr><td>⟨turn⟩</td><td>⟶</td><td>⟨diag⟩ ⟨turnradius⟩</td><td>1/4 turn[21f] starting in ⟨diag⟩</td></tr>
<tr><td></td><td>|</td><td>⟨cir⟩ ⟨turnradius⟩</td><td>explicit turn[21f]</td></tr>
<tr><td>⟨turnradius⟩</td><td>⟶</td><td>⟨empty⟩</td><td>use default turn radius</td></tr>
<tr><td></td><td>|</td><td><code>/</code> ⟨dimen⟩</td><td>set turnradius to ⟨dimen⟩</td></tr>
<tr><td>⟨segment⟩</td><td>⟶</td><td>⟨path-pos⟩ ⟨slide⟩ ⟨labels⟩</td><td>segment[21e] with ⟨slide⟩ and ⟨labels⟩</td></tr>
<tr><td>⟨slide⟩</td><td>⟶</td><td>⟨empty⟩ | <code><</code> ⟨dimen⟩ <code>></code></td><td>optional slide[21h]: ⟨dimen⟩ in the “above” direction</td></tr>
<tr><td>⟨labels⟩</td><td>⟶</td><td><code>^</code> ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩</td><td>label with ⟨it⟩[21i] above ⟨anchor⟩</td></tr>
<tr><td></td><td>|</td><td><code>_</code> ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩</td><td>label with ⟨it⟩[21i] below ⟨anchor⟩</td></tr>
<tr><td></td><td>|</td><td><code>|</code> ⟨anchor⟩ ⟨it⟩ ⟨alias⟩ ⟨labels⟩</td><td>break with ⟨it⟩[21j] at ⟨anchor⟩</td></tr>
<tr><td></td><td>|</td><td>⟨empty⟩</td><td>no more labels</td></tr>
<tr><td>⟨anchor⟩</td><td>⟶</td><td><code>-</code> ⟨anchor⟩ | ⟨place⟩</td><td>label/break placed relative to the ⟨place⟩ where <code>-</code> is a synonym for <code><>(.5)</code></td></tr>
<tr><td>⟨it⟩</td><td>⟶</td><td>⟨digit⟩ | ⟨letter⟩ | <code>{</code>⟨text⟩<code>}</code> | ⟨cs⟩</td><td>⟨it⟩ is a default label[21k]</td></tr>
<tr><td></td><td>|</td><td><code>*</code> ⟨object⟩</td><td>⟨it⟩ is an ⟨object⟩</td></tr>
<tr><td></td><td>|</td><td><code>@</code> ⟨dir⟩</td><td>⟨it⟩ is a ⟨dir⟩ectional</td></tr>
<tr><td>⟨alias⟩</td><td>⟶</td><td>⟨empty⟩ | <code>="</code>⟨id⟩<code>"</code></td><td>optional name for label object[21l]</td></tr>
</table>

Figure 14: ⟨path⟩s

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

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

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

21e. 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`.

21f. 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.

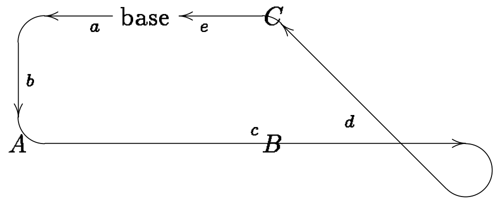
What the turn looks like is determined by the ⟨turn⟩ form:

⟨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.

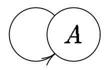
Note: Always write a valid ⟨pos⟩ after a ⟨turn⟩, otherwise any following ^ or _ labels can confuse the parser. So if you intend the `^r` in `` `^r `` to be a label then write `` `,^r ``, using a dummy `,` ⟨pos⟩ition.

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.

To Do: Curved turns using the new ellipsis feature.

21g. The last segment is exactly as a straight one except that the `>` action (if any) is executed (and cleared) just after the `<` action.

21h. "Sliding" a segment means moving each of the p, c objects in the direction perpendicular to the current direction at each.

21i. 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 21d 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.

21j. 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 21a and 21d.

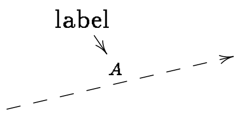
The special control sequence `\hole` is provided to make it easy to make an empty break.

21k. 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.

21l. 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



21.2 Arrows

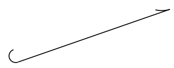
Arrows are paths with a particularly easy syntax for setting up arrows with *tail*, *stem*, and *head* in the style of [12]. This is provided by a single ⟨decor⟩ation the syntax of which is described in figure 15 (with the added convention that a raised '*' means 0 or more repetitions of the preceeding nonterminal).

Notes

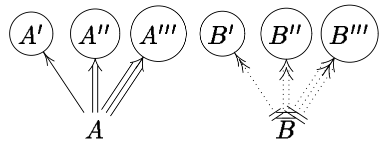
21m. Building an ⟨arrow⟩ is simply using the specified directionals (using `\dir` of §6.1) to build a path: the first ⟨tip⟩ becomes the *arrow tail* of the arrow, the ⟨conn⟩ection in the middle becomes the *arrow stem*, and the second ⟨tip⟩ becomes the *arrow head*. If a ⟨variant⟩ is given before the `{` then that variant `\dir` is used for all three. For example,

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

typesets



Exercise 25: Typeset these arrows:



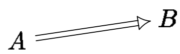
The above is a flexible scheme when used in conjunction with the kernel `\newdir` to define all sorts of arrowheads and -tails. For example,

```
\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

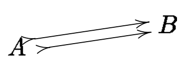


Notice that the fact that the directional uses only ⟨tipchar⟩ characters means that it blends naturally with the existing tips.

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



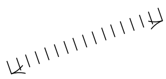
21n. Specifying a ⟨dir⟩ as a ⟨tip⟩ or ⟨conn⟩ means that `\dir`⟨dir⟩ is used for that ⟨tip⟩ or ⟨conn⟩. For example,

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

<table>
<tr><th colspan="2">Syntax</th><th>Action</th></tr>
<tr><td colspan="2">\ar ⟨arrow⟩ ⟨path⟩</td><td>make ⟨arrow⟩ along ⟨path⟩</td></tr>
<tr><td>⟨arrow⟩</td><td>⟶ ⟨form⟩*</td><td>⟨arrow⟩ has the ⟨form⟩s</td></tr>
<tr><td>⟨form⟩</td><td>⟶ @ ⟨variant⟩</td><td>use ⟨variant⟩ of arrow</td></tr>
<tr><td></td><td>| @ ⟨variant⟩ { ⟨tip⟩ }</td><td>build arrow[21m] using ⟨variant⟩ of a standard stem and ⟨tip⟩ for the head</td></tr>
<tr><td></td><td>| @ ⟨variant⟩ { ⟨tip⟩ ⟨conn⟩ ⟨tip⟩ }</td><td>build arrow[21m] using ⟨variant⟩ of ⟨tip⟩, ⟨conn⟩, and other ⟨tip⟩ as arrow tail, stem, and head (in that order)</td></tr>
<tr><td></td><td>| @ ⟨connchar⟩</td><td>change stem to the indicated ⟨connchar⟩</td></tr>
<tr><td></td><td>| @!</td><td>dash the arrow stem by doubling it</td></tr>
<tr><td></td><td>| @/ ⟨direction⟩ ⟨dist⟩ /</td><td>curve[21o] arrow the ⟨dist⟩ance towards ⟨direction⟩</td></tr>
<tr><td></td><td>| @(⟨direction⟩ , ⟨direction⟩)</td><td>curve to fit with in-out directions[21p]</td></tr>
<tr><td></td><td>| @‘ { ⟨control point list⟩ }</td><td>curve setup[21q] with explicit control points</td></tr>
<tr><td></td><td>| @[⟨shape⟩]</td><td>add [⟨shape⟩] to object ⟨modifier⟩s[21r] for all objects</td></tr>
<tr><td></td><td>| @* { ⟨modifier⟩* }</td><td>add object ⟨modifier⟩s[21r] for all objects</td></tr>
<tr><td></td><td>| @< ⟨dimen⟩ ></td><td>slide arrow[21s] the ⟨dimen⟩</td></tr>
<tr><td></td><td>| | ⟨anchor⟩ ⟨it⟩</td><td>break each segment at ⟨anchor⟩ with ⟨it⟩</td></tr>
<tr><td></td><td>| ^ ⟨anchor⟩ ⟨it⟩ | _ ⟨anchor⟩ ⟨it⟩</td><td>label each segment at ⟨anchor⟩ with ⟨it⟩</td></tr>
<tr><td>⟨variant⟩</td><td>⟶ ⟨empty⟩ | ^ | _ | 0 | 1 | 2 | 3</td><td>⟨variant⟩: plain, above, below, double, or triple</td></tr>
<tr><td>⟨tip⟩</td><td>⟶ ⟨tipchar⟩*</td><td>directional named as the sequence of ⟨tipchar⟩s</td></tr>
<tr><td></td><td>| ⟨dir⟩</td><td>any ⟨dir⟩ectional[21n]</td></tr>
<tr><td>⟨tipchar⟩</td><td>⟶ < | > | (|) | | | ’ | ‘ | + | /</td><td>recognised tip characters</td></tr>
<tr><td></td><td>| ⟨letter⟩ | ⟨space⟩</td><td>more tip characters</td></tr>
<tr><td>⟨conn⟩</td><td>⟶ ⟨connchar⟩*</td><td>directional named as the sequence of ⟨connchar⟩s</td></tr>
<tr><td></td><td>| ⟨dir⟩</td><td>any ⟨dir⟩ectional[21n]</td></tr>
<tr><td>⟨connchar⟩</td><td>⟶ - | . | ~ | = | :</td><td>recognised connector characters</td></tr>
</table>

Figure 15: ⟨arrow⟩s.

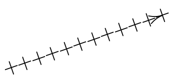
typesets



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
```

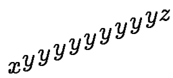
typesets



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

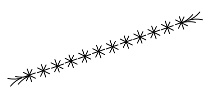
```
\xy\ar @{*{x}*{y}*{z}} (20,7)\endxy
```

typesets



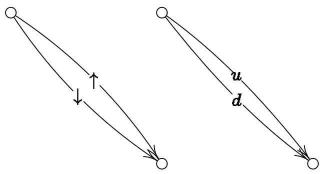
Note: A * introduces an ⟨object⟩ whereas the directional ‘•’ is typeset by the ⟨dir⟩ {*}.

Exercise 27: Typeset



using only one \ar command.

21o. *Curving* the arrow by */dℓ/*, 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

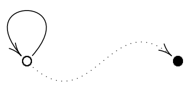
```
\xy
 \POS (0,10)  *\cir<2pt>{} ="a"
    , (20,-10)*\cir<2pt>{} ="b"
 \POS"a" \ar @/^1ex/ "b"|\uparrow
 \POS"a" \ar @/_1ex/ "b"|\downarrow
%
 \POS (20,10) *\cir<2pt>{} ="a"
    , (40,-10)*\cir<2pt>{} ="b"
 \POS"a" \ar @/u1ex/ "b"|u
 \POS"a" \ar @/d1ex/ "b"|d
\endxy
```

ℓ defaults to `.5pc` if omitted.

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

21p. Using `@(`d_2`,`d_2`)` where d_1, d_2 are ⟨direction⟩s (as described in note 4k) will typeset the arrow curved such that it leaves the source in direction d_1 and enters the target from direction d_2.

Exercise 28: Typeset



21q. The final curve form is the most general one: `@'{`⟨control point lists⟩`}` sets the control points explicitly to the ones in the ⟨control point lists⟩ (where they should be separated by `,`). See the curve extension described in §8 for the way the control points are used; when the control points list is parsed p is the source and c the target of the arrow.

21r. `@[...]` and `@*{...}` formations define 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; similarly if it is desired to make and entire arrow invisible then `@*{i}` can be used.

21s. `@<`D`>` will slide (each segment of) the arrow the dimension D as explained in note 21h.

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

22 Two-cell feature

Vers. 3.0 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.

22.1 Typesetting 2-cells in Diagrams

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.

Bug: This document still uses version 2 commands as described in appendix B.

$$\begin{tikzcd} A \arrow[r, bend left, "f"{name=U}] \arrow[r, bend right, "g"'{name=D}] \arrow[Rightarrow, from=U, to=D] & B \end{tikzcd}$$

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

$$\begin{tikzcd} A \arrow[r, bend left=50, "f"] \arrow[r, "g"'{pos=0.35}] \arrow[r, bend right=50, "h"'] \arrow[r, phantom, shift left=3, "\Downarrow\alpha"] \arrow[r, phantom, shift right=3, "\Downarrow\beta"] & B \end{tikzcd}$$

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

These categorical diagrams frequently have 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 16 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 §22.5.

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. Alternatively if the first character in the label is * then the label is set as an XY-pic ⟨object⟩, as if with \drop<object> or *<object> in the kernel language. The current direction is tangential to the curved arrows. Extra braces are needed to get a * as the label, as in ^{{{*}}} or _{{}*}.

The position of a label normal to the tangential direction can also 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. As above if the first character is * then the label is set as an XY-pic ⟨object⟩, the current direction being along the Arrow.

22.2 Standard Options

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 ...

_	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⟩. Also note that * has a special role when used as the first character; however it is considered to be part of the ⟨label⟩, see above.

22.3 Nudging

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

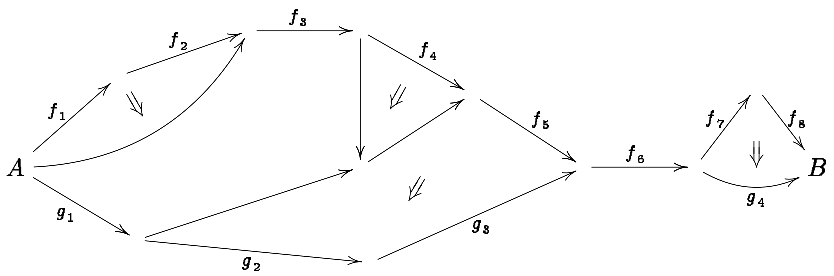


Figure 16: Pasting diagram.

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 16.
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*.

22.4 Extra Options

The following features are useful in non-mathematical applications.

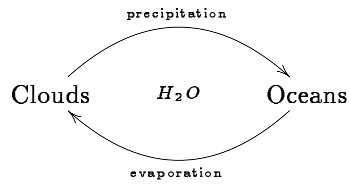
1. no Arrow

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.

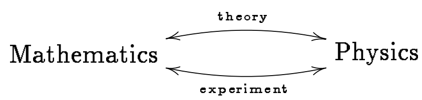
'	arrowheads pointing clockwise;
`	arrowheads pointing anti-clockwise;
"	arrow tips on both ends;
!	no tips at all.

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{>}`.




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




```
\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.

<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></td><td>|</td><td>{⟨tok⟩*⟨object⟩}</td><td>use ⟨object⟩ as the label</td></tr>
<tr><td></td><td>|</td><td>{⟨nudge⟩*⟨object⟩} | {*⟨object⟩}</td><td></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></td><td>|</td><td>*⟨object⟩ | ⟨nudge⟩*⟨object⟩</td><td>set ⟨object⟩, displaced by ⟨nudge⟩</td></tr>
<tr><td>⟨nudge⟩</td><td>⟶</td><td><⟨number⟩></td><td>use ⟨number⟩ in an appropriate way, e.g., to position object or label along a fixed axis</td></tr>
<tr><td></td><td>|</td><td><\omit></td><td>do not typeset the object/label</td></tr>
</table>

Figure 17: ⟨twocell⟩s

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*.

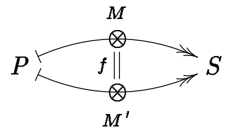
These effects are specified by placing options after the \..twocell control sequence, *e.g.* \rtwocell*options labels*... . Each option is either a single token ⟨tok⟩, or a ~⟨tok⟩ with a single argument: ~⟨tok⟩{*arg*}. Possibilities are listed in the following table, in which {..} denotes the need for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




```
\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.

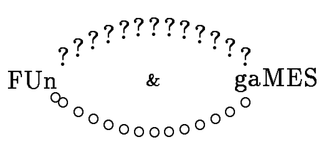
These specifications apply to all 2-cells subsequently constructed at the same level of TEX grouping. Alternatively using a ~-option, as in *Extra Options 2*, allows such a specification for a single 2-cell or curved part.

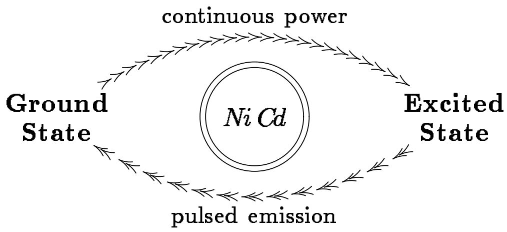
Objects used to construct curves can be of two types. Either a single ⟨object⟩ is set once, with copies placed along the curve. Alternatively a directional object can be aligned with the tangent along the curve. In this case use a specification takes the form:

\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.

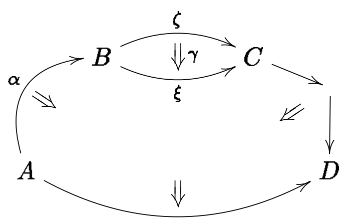




22.5 2-cells in general XY-pictures

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
```



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{}},
"B";"C",{\twocell^\zeta_\xi{\gamma}},
"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.

23 Matrix feature

Vers. 3.0 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 §23.1, then in §23.2 we summarise the new ⟨coord⟩inate forms used to refer to entries. In §23.3 we explain what parameters can be set to change the spacing and orientation of the matrix, and in §23.4 we explain how the appearance of the entries can be changed.

23.1 XY-matrices

The fundamental command of this feature is

`\xymatrix` ⟨setup⟩ `{`⟨rows⟩`}`

---

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 `&` (we discuss in the following sections what ⟨setup⟩ can be). 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

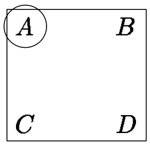
- arbitrary XY-pic ⟨decor⟩ations may be specified in each entry and will be interpreted in a state where c is the current entry,

- 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 §23.4)[10], so use `{*}` to get a `*`.

For example,

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

will typeset



Matrices are often quite slow to typeset so as a convenience all matrices can be set to compile (and not) automatically with the declarations

```
\CompileMatrices
\NoCompileMatrices
```

Individual matrices can be compiled or not using the explicit commands `\xymatrixcompile` and `\xymatrixnocompile` as well as by encapsulating in the usual `\xycompileto{`*name*`}{...}` (see note 5e).

Bug: Matrices will only compile correctly if all entries start with a nonexpandable token, *i.e.*, `{` or `\relax` or some non-active character. Matrix nesting is not safe.

23.2 New coordinate formats

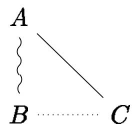
It is possible within entries to refer to all the entries of the XY-matrix using the following special ⟨coord⟩inate forms:

`"`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

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

So the current entry has the synonyms `[0,0]`, `[]`, `[rl]`, `[ud]`, `[dudu]`, etc., as well as its 'absolute' name `"`r`,`c`"`.

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



was typeset by

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

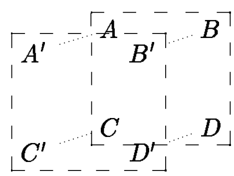
If an entry outside the XY-matrix is referenced then an error is reported.

In case several matrices are used in the same diagram, and they refer to each other, then it is useful to give the matrices different `"`⟨prefix⟩`"` ⟨setup⟩ such that they can refer to each other using the following special coordinate forms that all have the same meaning except the target entry is picked from a particular matrix:

`"`⟨prefix⟩r`,`c`"`
`["`⟨prefix⟩`"`Δr`,`Δc`]`
`["`⟨prefix⟩`"` ⟨hop⟩* `]`

In fact absolute references *must* be given as `"`⟨prefix⟩⟨row⟩`,`⟨col⟩`"`, *even inside the matrix itself*.

Here is an example using this:



was typeset (using the 'frame' extension and 'arrow' feature) by

```
\xy
  \xymatrix"*"{%
   A & B \\
   C & D }%
  \POS*\frm{--}
  \POS-(10,3)
  \xymatrix{%
   A' \ar@{.}["*"] & B' \ar@{.}["*"] \\
   C' \ar@{.}["*"] & D' \ar@{.}["*"] }%
  \POS*\frm{--}
\endxy
```

23.3 Spacing and rotation

Any matrix can have its spacing and orientation changed by adding setup 'switches' between `\xymatrix` and the opening `{`.

The default spacing for a matrix is changed with the switches

---

`@R`⟨add op⟩ ⟨dimen⟩
`@C`⟨add op⟩ ⟨dimen⟩
`@` ⟨add op⟩ ⟨dimen⟩

---

that change row spacing, column spacing, and both, respectively, as indicated by the ⟨add op⟩ and ⟨dimen⟩, where the ⟨dimen⟩ may be omitted and can be given as `R` or `C` to indicate the current values of row or column spacing, but there is *no default*.

In addition, XY-pic can be instructed to 'fix the grid' of the matrix with the switches

---

`@!R`
`@!C`
`@!`

---

that ensure that the row spacing, column spacing, and both, of *all* entries is as if they had all been the size of the largest entry.

The spacing can also be changed for en entire TEX group by the declarations

---

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

The default spacing for both is 2pc.

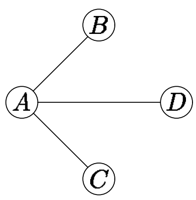
An entire matrix can be rotated by adding a rotation ⟨setup⟩ of the form

---

`@`⟨direction⟩

This will set the orientation of the rows to ⟨direction⟩ (the default corresponds to `r`, *i.e.*, rows are oriented left to right).

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.

23.4 Entries

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

---

`*` ⟨object⟩ ⟨pos⟩ ⟨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.

Additional object ⟨modifier⟩s may be added to an otherwise ordinary entry by using the forms

---

`**[`⟨shape⟩`]` ⟨entry⟩
`**{`⟨modifier⟩`*}` ⟨entry⟩

---

this is useful for recentering entries.

Exercise 32: Typeset the following diagram:

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

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⟩ `}`

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

---

`\everyentry={` ⟨decor⟩ `}`

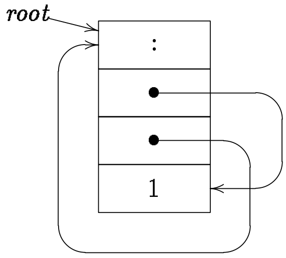
---

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.

24 Graph Combinator feature

Vers. 3.0 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 [5] 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 makes use of facilities of the 'arrow' feature option which is therefore required.

Figure 18 summarises the syntax of a ⟨graph⟩ with notes below. A ⟨graph⟩ can appear either in an XY-picture (as ⟨decor⟩) or "stand-alone".

Notes

24a. A *move* is to establish a new *current node*.

24b. 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 §21 can be used, in particular arrows can be labelled and segmented, but with the change that ⟨path-pos⟩ means ⟨node⟩ as explained in note §21e.

24c. 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.

24d. The `&` and `\\` special moves are included to make it simple to enter 'matrix-like' things as graphs – note that they will not be automatically aligned, however, for that you should use the `!M` escape.

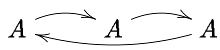
`&` is the same as `[r]` and `\\` is the same as `[r]!{y+(0,-1)-(0,0)}` which uses a kernel escape to moves to the first column in the next row (where the first column is on the y-axis of the current coordinate system).

Note: If you use the form `*{...}` for nodes then you don't have to change them if you decide to use an XY-matrix.

24e. 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?



24f. 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 the current base is redefined using `[]!{...}` with an appropriate ⟨pos⟩ition containing `:` and `::` as described in note 3d.

To Do: Describe the use of ⟨move⟩s with ⟨place⟩s in detail ... in particular (1) 'until perpendicular to ...' and (2) 'until intercepts with ...' can be coded...

24g. 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` ⟨pos⟩ ⟨decor⟩ `\restore}`.

24h. It is possible to insert a ⟨matrix⟩ in a graph provided the 'matrix' option described in §23 has been loaded: it overwrites the node with the result of `\xymatrix`⟨matrix⟩. Afterwards the graph 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.

Bug: `[dr]` immediately after the matrix will work as expected, *e.g.*, make the center of `"2,2"` the

Syntax			Action
\xygraph{⟨graph⟩}			typeset ⟨graph⟩
⟨graph⟩	⟶	⟨step⟩*	interpret ⟨step⟩s in sequence
⟨step⟩	⟶	⟨node⟩	move[24a] to the ⟨node⟩
	|	-⟨arrow⟩ ⟨node⟩ ⟨labels⟩	draw[24b] line to ⟨node⟩, with ⟨labels⟩
	|	:⟨arrow⟩ ⟨node⟩ ⟨labels⟩	draw[24b] ⟨arrow⟩ to ⟨node⟩, with ⟨labels⟩
	|	(⟨list⟩)	map[24c] current node over ⟨list⟩
⟨node⟩	⟶	[⟨move⟩]	new node ⟨move⟩d relative to current
	|	& | \\	new node in next column/row[24d]
	|	"⟨id⟩"	previously saved[24e] node
	|	?	currently mapped[24c] node
	|	⟨node⟩ ⟨it⟩	⟨node⟩ with ⟨it⟩ typeset and saved[24e] there
	|	⟨node⟩ = "⟨id⟩"	⟨node⟩ saved[24e] as "⟨id⟩"
	|	⟨node⟩ ! ⟨escape⟩	augment node with material in another mode
⟨move⟩	⟶	⟨hop⟩*	⟨hop⟩s[24f] (dulr) from current node
	|	⟨hop⟩* ⟨place⟩ ⟨move⟩	do ⟨hop⟩s[24f] but use its ⟨place⟩ and ⟨move⟩ again
⟨list⟩	⟶	⟨graph⟩ , ⟨list⟩ | ⟨graph⟩	list of subgraphs[24c]
⟨escape⟩	⟶	{ ⟨pos⟩ ⟨decor⟩ }	perform ⟨pos⟩ ⟨decor⟩[24g]
	|	M ⟨matrix⟩	insert ⟨matrix⟩[24h]
	|	P ⟨polygon⟩	insert ⟨polygon⟩[24i]

Figure 18: ⟨graph⟩s

current node, but others might not, *e.g.*, [rr] will not necessarily place the current node on top of "1,3".

24i. It is possible to insert a ⟨polygon⟩ in a graph provided the poly option described in §25 has been loaded: it will have its center on top of the current node and default radius as the ⟨hop⟩ base size.

It is possible to add new graph escapes of the form !⟨letter⟩ with the command

\newgraphescape{⟨letter⟩}{⟨graph⟩}

that makes the specified escape generate the ⟨graph⟩ as a macro.

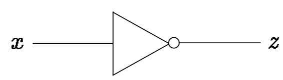
Note: It is possible to pass arguments to the ⟨graph⟩ using the standard TEX \def method: The declaration code

```
\newgraphescape{i}#1#2{
 []!{+0="o#2"*=<10pt>{};p!#1**{},"o#2"
  -/4pt/*!E\cir<2pt>{}
  +0;p-/:a(-30)24pt/**\dir{-}="X2"
  ;p-/:a(-60)24pt/="X1"**\dir{-};?(.5),="i#2",
  p-/:a(-60)24pt/**\dir{-},
  "o#2"."i#2"."X1"."X2"}}
```

is (rather complicated kernel code) that makes the node escape !i*dn* typeset an 'inverter' oriented with the *d* corner as the output with input named "i*n*" and output named "o*n*" such that the graph

```
\xygraph{ []!iR1 ("i1"[l]x - "i1") - [r]z }
```

will typeset



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   ( ? :@/^/   ^x        "X" ,
          ? :@{-->} |-{(x,y)} "xy" ,
          ? :@/_/   _y        "Y" )   }
```

typesets

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

25 Polygon feature

Vers. 3.0 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 any number (≥ 3) of 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⟩...`}`

---

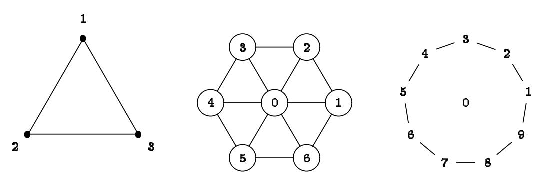
where the `"`⟨prefix⟩`"` and ⟨switches⟩ are optional. Their uses will be described shortly.

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.

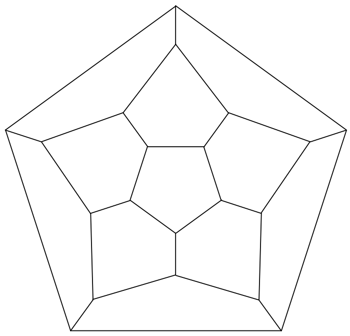
The macro `\xypolyname` holds the name of the polygon, which is ⟨prefix⟩ if supplied. In this case the value of `\xypolynum` is also stored as `\`⟨prefix⟩`NUMSIDES`, accessible outside the polygon.

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.



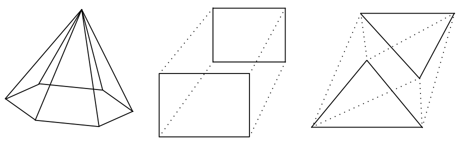
Exercise 35: Give code to typeset these.

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
```

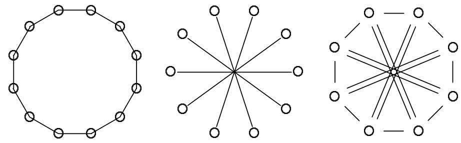
Next a hexagonal pyramid, a rectangular box and an octahedral crystal specified as a triangular anti-prism. Notice how the ~:{...} switch is used to create non-square bases, allowing the illusion of 3D-perspective in the resulting diagrams:



```
\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.



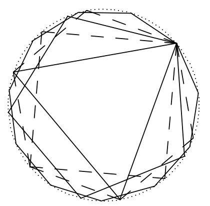
```
\xy/r2pc/: {\xypolygon12{\circ}},
+/r5pc/,{\xypolygon10{~<{-}~>{}{\circ}}},
+/r5pc/,{\xypolygon8{~*{\circ}~<=}}\endxy
```

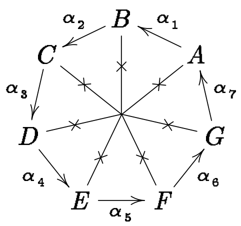
Switches
The allowable switches are given in the following table:

~:{...}	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 15, 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 14.

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:

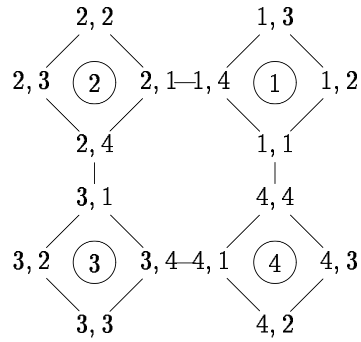
```
Xy-pic Warning:  vertex already specified,
         discarding unused tokens:
```

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

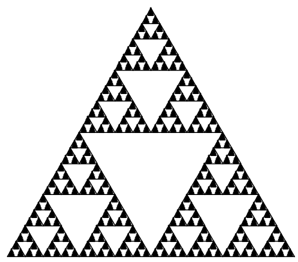
When `\xypolygon` is specified within a `~<>{...}` or `~>>{...}` switch for another polygon, then the inner polygon inherits a name which incorporates the number of the part on which it occurs, as given by `xypolynode`. This name is accessed using `\xypolyname`. In the following example the inner polygon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




```
\xypolygon4{~:{/r4pc/:}
 ~<>{*\frm<8pt>{o}\xypolygon4{~:{/-2pc/:}
   ~*{\xypolyname\xypolynode}}}
 [o]=<5pc>{\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.



```
\font\msamv=msam5 at 5pt
\def\blacktriangle{\hbox{\msamv\char'116}}
\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.

26 Lattice and web feature

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{web}

This feature provides macros to facilitate typesetting of arrangements of points within a 2-dimensional lattice or "web-like" structure.

Currently the only routines implemented with this feature are some "quick and dirty" macros for dropping objects at the points of an integer lattice. **To Do:** More sophisticated routines will be developed for later versions of XY-pic, as the need arises.

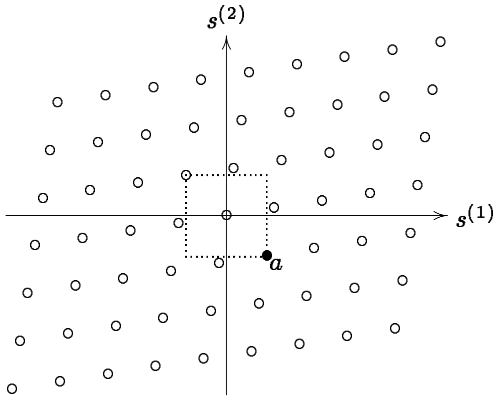
Mathematically speaking, let $\vec{u}$ and $\vec{v}$ be vectors pointing in independent directions within the plane. Then the lattice spanned by $\vec{u}$ and $\vec{v}$ is the infinite set of points L given by:

$$L = \left\{ a\,\vec{u} + b\,\vec{v}\,;\ \text{for } a,\, b \text{ integers} \right\}.$$

Within XY-pic the vectors $\vec{u}$ and $\vec{v}$ can be established as the current coordinate basis vectors. The following macros typeset a finite subset of an abstract lattice.

`\xylattice#1#2#3#4`	points in lattice
`\croplattice#1#2#3#4#5#6#7#8`	...in specific rectangle.

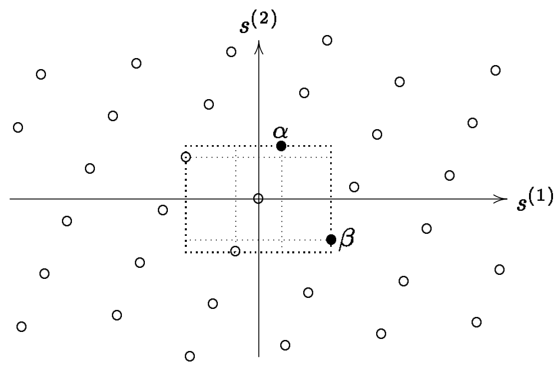
The parameters #1 ...#4 are to be integers $a_{\min}$, $a_{\max}$, $b_{\min}$ and $b_{\max}$, so that the portion of the lattice to be typeset is that collection of vectors in L for which $a_{\min} \le a \le a_{\max}$ and $b_{\min} \le b \le b_{\max}$.




```
\def\latticebody{%
\ifnum\latticeA=1 \ifnum\latticeB=-1 %
\else\drop{\circ}\fi\else\drop{\circ}\fi}
\xy *\xybox{0;<1.5pc,1mm>:<1mm,1.5pc>::
 0,{\xylattice{-4}4{-3}3}
 ,(1,-1)="a"*{\bullet}*+<2pt>!UL{a}
 ,(-1,1)."a"*\frm{.}="L"
,{"L"+L \ar "L"+R*+!L{s^{(1)}}}
,{"L"+D \ar "L"+U*+!D{s^{(2)}}}
\endxy
```

In the above code, notice how the basis is first established then the \xylattice typeset. Doing this within an \xybox allows axes to be sized and placed appropriately. Since lattice points are determined by their (integer) coordinate displacements, they can be revisited to add extra ⟨object⟩s into the overall picture. More generally, the origin for lattice-coordinates is the current ⟨pos⟩ c, when the \xylattice command is encountered. Easy accessibility is maintained, as seen in the next example.

When the basis vectors $\vec{u}$ and $\vec{v}$ are not perpendicular the collection of points with a, b in these ranges will fill out a skew parallelogram. Generally it is useful to plot only those points lying within a fixed rectangle. This is the purpose of \croplattice, with its extra parameters #5 ...#8 determining the 'cropping' rectangle within which lattice points will be typeset. Other points will not be typeset even when a and b are within the specified ranges. Explicitly the horizontal range of the cropping rectangle is $X_{\min}$ to $X_{\max}$, with $X_{\min}$ being the X-coordinate of the vector #5 $\times\,\vec{u}$, where #5 is a ⟨number⟩ (not necessarily an integer). Similarly $X_{\max}$ is the X-coordinate of #6 $\times\,\vec{u}$. The vertical extents are $Y_{\min}$ and $Y_{\max}$, given by the Y-coordinates of #7 $\times\,\vec{v}$ and #8 $\times\,\vec{v}$ respectively.




```
\def\latticebody{%
\ifnum\latticeA=1 \ifnum\latticeB=-1 %
\else \drop{\circ}\fi\else
\ifnum\latticeA=0 \ifnum\latticeB=1\else
\drop{\circ}\fi\else\drop{\circ}\fi\fi}
\xy +(2,2)="o",0*\xybox{%
 0;<3pc,1.5mm>:<0.72pc,1.65pc>::,{"o"
 \croplattice{-4}4{-4}4{-2.6}{2.6}{-3}3}
 ,"o"+(0,1)="a"*{\bullet}*+!D{\alpha}
```

```
 ,"o"+(1,-1)="b"*{\bullet}*+!L{\beta}
 ,"o"+(0,-1)="c","o"+(-1,1)="d"
 ,"a"."c"="e",!DR*{};"a"**\dir{.}
 ,"e",!UL*{};"c"**\dir{.}
 ,"b"."d"="f",!DL*{};"b"**\dir{.}
 ,"f",!UR*{};"d"**\dir{.}
 ,"e"."f"*\frm{.}="L","o"."L"="L"
 ,{"L"+L \ar "L"+R*+!L{s^{(1)}}}
 ,{"L"+D \ar "L"+U*+!D{s^{(2)}}}
\endxy
```

The `\latticebody` macro. At each lattice point within the specified range for a, b (and within the cropping rectangle when `\croplattice` is used), a macro called `\latticebody` is expanded. This is meant to be user-definable, so as to be able to adapt to any specific requirement. It has a default expansion given by ...

```
\def\latticebody{\drop{\bullet}}.
```

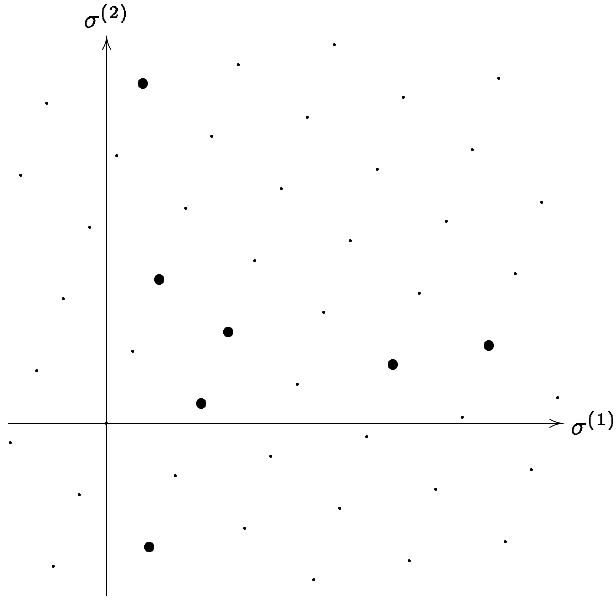
The following macros may be useful when specifying what to do at each point of the lattice.

`\latticebody`	expanded at lattice points
`\defaultlatticebody`	resets to default
`\latticeA`	a-value of lattice point
`\latticeB`	b-value of lattice point
`\latticeX`	X-coord, offset in pts...
`\latticeY`	Y-coord, ...from lattice origin.

As in the examples presented above, the object dropped at the lattice point can be varied according to its location, or omitted altogether.

In the final example the `\latticebody` macro performs a calculation to decide which lattice points should be emphasised:

```
\def\latticebody{\dimen0=\latticeX pt
 \ifdim\dimen0>0pt \divide\dimen0 by 64
 \dimen0=\latticeY\dimen0 \relax
 \ifdim 0pt>\dimen0 \dimen0=-\dimen0 \fi
 \ifdim 10pt>\dimen0 \drop{\bullet}%
 \else\drop{.}\fi \else\drop{.}\fi}
\xy*\xybox{0;<3pc,2.57mm>:<.83pc,2.25pc>::
,{\croplattice{-3}5{-5}5
  {-1.3}{4.5}{-3.4}{4.4}}="L"
,{"L"+L \ar "L"+!R*+!L{\sigma^{(1)}}}
,{"L"+D \ar "L"+!U*+!D{\sigma^{(2)}}}
\endxy
```



27 Circle, Ellipse and Arc feature

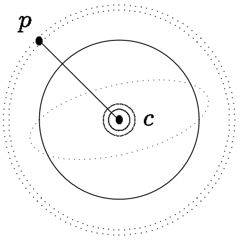
Vers. ? by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: `\xyoption{arc}`

This feature provides a means to a specify circles of arbitrary radius, drawn with a variety of line styles. Similarly ellipses may be specified, having arbitrary major/minor axes aligned in any direction. A circular arc joining two points can be constructed with specified tangent direction at one end.

All the curves described here—circles, ellipses and sectors of these—are constructed using the curves from the `xycurve` extension. As such any comments given there concerning memory requirements are equally valid here, perhaps even more so. Use of the `xyps` PostScript back-end is highly recommended.

27.1 Full Circles

The `xyarc` feature allows a much wider range of possibilities for typesetting circles than is available with `\cir`. Firstly the radius is no longer restricted to a finite collection of sizes. Secondly fancy line (curve) styles are available, as with curved arrows. Finally there are a variety of ways of specifying the desired radius, relative to other parts of the picture being built, as in the following example.




```
\xy 0;/r5pc/:*\dir{*}="p",*+!DR{p};
p+(.5,-.5)*\dir{*}="c",*+++!L{c}**\dir{-}
,{\ellipse<>{:}},{\ellipse(.5){}}
,0;(.5,.5)::,"p";"c",{\ellipse(.5){.}}
,{\ellipse<5pt>{=}}\endxy
```

The following give circles centred at c.

`\ellipse<>{`⟨style⟩`}`	radius = dist(p,c)
`\ellipse<`⟨dimen⟩`>{..}`	radius is the ⟨dimen⟩
`\ellipse(`⟨num⟩`){`⟨style⟩`}`	unit circle scaled ⟨num⟩, in the current basis.

Note that if the current basis is not square then the latter variant, namely `\ellipse(`⟨num⟩`)`, will typeset an ellipse rather than a circle. On the other hand the first two variants always specify true circles. In the 2nd case, i.e. when ⟨dimen⟩ is ⟨empty⟩, the size of the object at p is taken into account when drawing the circle; if this is not desired then kill the size using a null object, e.g. `;*{};`.

Currently the `\ellipse` macro works only as a ⟨decor⟩. In future versions there will be an ⟨object⟩ called `\arc` which will have elliptical shape, via `\circleEdge` with possibly unequal extents. Also it will be possible to `\connect\arc`, which will set the current connection so that any place on the full ellipse, not just the visible sector, will be accessible using an extension to the usual ⟨place⟩ mechanism.

To Do: make this be!!

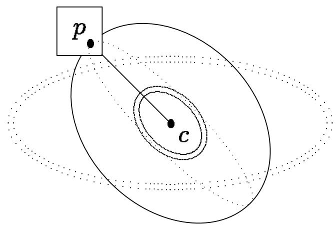
27.2 Ellipses

There are several ways to specify an ellipse, apart from the method illustrated above in which the basis must be changed from square. Basically we must specify the lengths of the major and minor axes. Also it is necessary to specify an alignment for one axis.

In the following, the ellipse is centred on c and one axis is aligned along the line $\overline{pc}$, except with the final variant where it aligns with the current basis. When used ⟨num⟩ is treated as a scale factor, multiplying an appropriate length.

`\ellipse<`⟨dimen⟩`,`⟨dimen⟩`>{..}`	given axes lengths
`\ellipse<,`⟨dimen⟩`>{`⟨style⟩`}`	one axis is $\overline{pc}$
`\ellipse(,`⟨num⟩`){`⟨style⟩`}`	...perp axis is scaled
`\ellipse(`⟨num⟩`,`⟨num⟩`){..}`	scaled axes aligned with basis.

In the latter variant, if the second ⟨num⟩ is ⟨empty⟩ then this is equivalent to both ⟨num⟩s having the same value, which is in turn equivalent to the final variant for circles.




```
\xy 0;/r5pc/:*\dir{*},*++!DR(.5){p}
*\frm{-};p+(.5,-.5)*\dir{*}="c",
**\dir{-},*+!UL{c},"c",
,{\ellipse(1,.4){:}},{\ellipse(,.75){}}
,{\ellipse<15pt,10pt>{=}}
;*{};{\ellipse<,10pt>{.}}\endxy
```

27.3 Circular and Elliptical Arcs

The `xyarc` feature handles arcs to be specified in two essentially different ways, according to what information is provided by the user. We call these the "radius-unknown/end-points known" and the "radius-known/end-points unknown" cases.

radius unknown, end-points known

The simplest case, though not necessarily the most common, is that of a circular arc from p to c, with radius and centre unspecified. To uniquely specify the arc, the tangent direction at p is taken to be along the current direction, given by `\Direction`, as set by the latest ⟨connect⟩ion. If no connection has been used, then the default ⟨direction⟩ is "up".

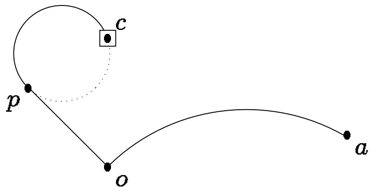
`\ellipse_{`⟨style⟩`}`	clockwise arc from p to c
`\ellipse^{`⟨style⟩`}`	counter-clockwise arc
`\ellipse{`⟨style⟩`}`	also counter-clockwise

With this information only, a unique circle can be found whose radius and centre need not be specified in advance. For a unique arc it is sufficient to specify the orientation around the circle.

The exception is when the current direction is from p to c, in which case no circle exists. Instead a straight line is typeset accompanied by the following message:

```
Xy-pic Warning: straight arc encountered
```

The following example shows how, given three points o, p and c, to continue the line $\overline{op}$ by a circular arc to c joining smoothly at p.




```
\xy 0;/r5pc/:*=+\dir{*}*+!UR{p};
p+(.5,-.5)*\dir{*}="o",*+!UL{o}
,+(0,.81)*=<6pt>\dir{*}*\frm{-}="c"
,*+!DL{c},"o",**\dir{-},
"c",{\ellipse_{}},{\ellipse^{.}}
%
,"o"+(1.5,.2)*\dir{*}="a"*+!UL{a}
,"o";p+/_1pc/,**{},"a",{\ellipse_{}}
\endxy
```

Note how the remainder of the circle can be specified separately. The example also shows how to specify an arc which leaves a particular point perpendicular to a specific direction.

Slightly more complicated is when the tangent direction at p is specified, but different from the current direction; a unique circular arc can still be defined. More complicated is when a specific tangent direction is required also at c. In this case the arc produced is a segment of an ellipse. (If the required tangent at p points to c then a straight segment is drawn, as in the circular case described above.)

`\ellipse`⟨dir⟩$_p$`,`⟨orient⟩`{..}`	circular
`\ellipse`⟨dir⟩$_p$`,`⟨orient⟩`,`⟨dir⟩$_c$`{..}`	elliptical
`\ellipse`⟨dir⟩$_p$`,`⟨orient⟩⟨dir⟩$_c$`{..}`	elliptical
`\ellipse`⟨dir⟩$_p$`,`⟨orient⟩`,=`⟨dir⟩$_c$`{..}`	elliptical
`\ellipse`‘⟨coord⟩⟨orient⟩`{..}`	elliptical

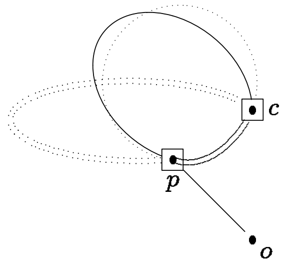
In these cases ⟨dir⟩$_p$ and ⟨dir⟩$_c$ are ⟨direction⟩ specifications, as in figure 3 and note 4k, and ⟨orient⟩ must be either `^` or `_` for anti-/clockwise respectively, defaulting to `^` if ⟨empty⟩. Beware that the `(*`⟨pos⟩⟨decor⟩`*)` form *must* be used for this ⟨direction⟩ variant, as if an object modifier.

The second and third cases in the above table generally give identical results. The second ‘`,`’ is thus optional, except in two specific situations:

1. ⟨orient⟩ is empty and ⟨dir⟩$_c$ has `^` or `_` as the first token;

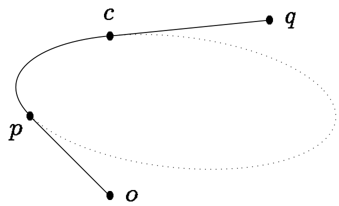
2. ⟨orient⟩ is `^` and ⟨dir⟩$_c$ has `^` as first token. Without the `,` then `^^` would be interpreted by TEX as part of a special ligature for a hexadecimal character code.

If both ⟨orient⟩ and ⟨dir⟩$_c$ are ⟨empty⟩ then even the first ‘`,`’ can be omitted.




```
\xy 0;/r5pc/:*=<8pt>\dir{*}="p",*\frm{-}
,*++!U{p},"p";p+(.5,-.5)*+\dir{*}="o"
,*+!UL{o},+(0,.81)*=<8pt>\dir{*}="c"
,*\frm{-},*++!L{c},"o"**\dir{-},"c"
,{\ellipse :a(50),_:0{:}}
,{\ellipse :a(30),_:a(-45){}}
,{\ellipse :a(40),_{.}},
;*{};{\ellipse :a(20),^=_{=}}\endxy
```

Note that only the slope of ⟨dir⟩$_p$ and ⟨dir⟩$_c$ is significant; rotations by 180° being immaterial.



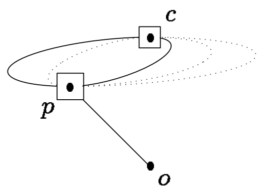

```
\xy 0;/r5pc/:*\dir{*}="p",*+!UR{p}
;p+(.5,-.5)*\dir{*}="o",*++!L{o}**\dir{-}
,p+(.5,.5)*\dir{*}="c",*++!D{c},"c"
;p+(1,.1)*\dir{*}="q",*++!L{q}**\dir{-}
,"o";"p",**{};"c"
,{\ellipse![["o";"p"]],_![["q";"c"]]{}}
,{\ellipse![["o";"p"]],![["c";"q"]]{.}}
\endxy
```

The = variant establishes the ⟨direction⟩ parsing to begin with the direction resulting from ⟨dir⟩$_p$ instead of the original direction. If ⟨dir⟩$_c$ is required to be the original direction then use `:0`. It cannot be ⟨empty⟩ since this is interpreted as requiring a circular arc with unspecified tangent at c; see the example above. However when ⟨dir⟩$_p$ and ⟨dir⟩$_c$ are parallel there is a whole family[11] of possible ellipses with the specified tangents.

With no further hint available, a choice is made based on the distance between p and c. If the required direction is perpendicular to $\overline{pc}$ this choice results in a circular arc. The optional factor in `=(`⟨num⟩`)` is used to alter this choice; the default `(1)` is assumed when nothing follows the =. This factor is used to "stretch"

[11] Indeed this is always so. The algorithm used for the general case tends toward parallel lines—clearly unsuitable.

the ellipse along the specified direction. For a negative ⟨num⟩ the orientation reverses.




```
\xy ;/r5pc/:*+=<10pt>\dir{*}="p";p*\frm{-}
,*++!UR{p},p+(.5,-.5)*\dir{*}="o",**\dir{-}
,*+!UL{o},+(0,.81)*=<8pt>\dir{*}="c"
,*\frm{-},*++!DL{c},"c"
,{\ellipse r,={}},{\ellipse r,=(2){.}}
,{\ellipse r,^=(3){.}},{\ellipse r,=(-2){}}
,{\ellipse r,=(-1){.}}\endxy
```

The final variant uses the directions from p and c to the given ⟨coord⟩. If ⟨orient⟩ is ⟨empty⟩ then the orientation is determined to give the shortest path along the ellipse. Specifying an ⟨orient⟩ of `^` or `_` will force the orientation, even if this means travelling 'the long way' around the ellipse. For an example, see the next figure.

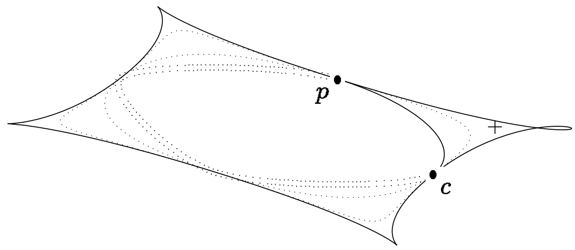
Alternative curves In some cases the circular/elliptic curve can be replaced by a curve with a different shape, having the same tangent directions at the end-points. When a full circle/ellipse is specified then one gets instead a closed curve constructed from 4 spline segments. Other variants use a single segment, 2 or 3 segments, or some portion of all 4 segments. Possibilities are given in the following table.

`\ellipse~e ...{`⟨..⟩`}`	elliptical, as above
`\ellipse~q ...{`⟨..⟩`}`	parabolic segments
`\ellipse~c ...{`⟨..⟩`}`	cubic segments
`\ellipse~i ...{`⟨..⟩`}`	interpolating cubic
`\ellipse~p ...{`⟨..⟩`}`	cuspidal cubic
`\ellipse~c(`⟨num⟩`) ...{`⟨..⟩`}`	cubic segments, with "looseness"

In the latter case the ⟨num⟩, typically between 0 and 1, controls how soon the curve begins to bend away from the tangent direction. Smaller values give a tighter curve — 0 gives a straight line — with `~c` being the same as `~c(1)` and `~q` is `~c(.66667)`, that is ⟨num⟩$=\frac{2}{3}$.

The curve produced by the "interpolating" variant `~i` actually passes through the control point `"x"`, with slope parallel to the line $\overline{pc}$. Since the tangents at p and c point toward `"x"` the curvature is quite gentle until near `"x"` where the curve bends rapidly, yet smoothly. This is obtained also by using `~c(1.33333)`, that is ⟨num⟩$=\frac{4}{3}$. Since `<num>` > 1 the "convex hull property" does not hold; indeed the curve is entirely outside the convex hull of p, c and `"x"`, apart from those points themselves.

The 'cuspidal' variant `~p` is equivalent to `~c(2)`. It exhibits a cusp. For `<num>` > 2 the curve is so "loose" that it exhibits loops. (The author offers no guarantees on the usefulness of such curves for any particular purpose; however they do look nice. ☺)




```
\xy 0;/r6pc/:*+\dir{*}="p",*+!UR{p},"p";
p+(.5,-.5)*+\dir{*}="c",*+!UL{c}
,"p"+(.825,-.25)="x"*\dir{+},"c"
,{\xycompile{\ellipse`"x"{-}}}
,{\xycompile{\ellipse~q`"x"^{.}}}
,{\xycompile{\ellipse~c`"x"{.}}}
,{\xycompile{\ellipse~c(.3)`"x"^{:}}}
,{\xycompile{\ellipse~c(2.3)`"x"{-}}}
,{\xycompile{\ellipse~i`"x"^{.}}}
,{\xycompile{\ellipse~p`"x"^{-}}}
\endxy
```

Hint: When exploring to find the best location for the "control-point" (e.g. the `"x"` in the above example), then use `\xycompile` as shown, changing the location outside of the compilation. This speeds up the reprocessing with the changed value.

Avoiding overflows If $\langle\text{dir}\rangle_p$ and $\langle\text{dir}\rangle_c$ are intended to be equal then the method of the previous paragraph should be used. However it may happen that "nearly parallel" directions may be specified, perhaps by accident. There is then the possibility of "numerical overflow" or a "division by zero" error. The latter may be accompanied by a warning message:

```
Xy-pic Warning: division by 0 in
     \intersect@, replaced by 50
```

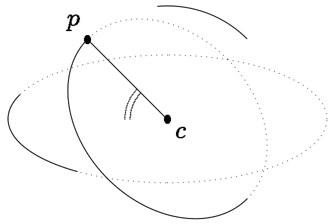
This indicates that the number 50 has been used as the result of a division by zero. In many contexts this will produce an acceptable result. However it may lead to an "overflow" in other situations, or to drawing beyond the normal page boundary. This can be controlled using a ⟨decor⟩ of type `,{\zeroDivideLimit{`⟨num⟩`}}`, prior to specifying the `\ellipse`. The value 50 will be replaced by ⟨num⟩ whenever a "division by zero" would otherwise be encountered in an intersection calculation.

radius known, end-points unknown

The language for these is a combination of most of that used above, but the interpretation of the ⟨direction⟩s is different...

```
\ellipse<⟨radius⟩>⟨dir⟩₁,⟨orient⟩,⟨dir⟩₂{..}
\ellipse<⟨radius⟩>⟨dir⟩₁,⟨orient⟩,=⟨dir⟩₂{..}
```

where ⟨radius⟩ is one of the forms used above to describe a circle or ellipse. Not all of the ellipse will be typeset—only that arc starting with $\langle\text{dir}\rangle_1$ as tangent vector, tracing via ⟨orient⟩ until the tangent points in direction $\langle\text{dir}\rangle_2$. This effectively extends the notation used with \cir in 6.2. Note that rotating a given $\langle\text{dir}\rangle_i$ by 180° specifies a different arc on the same ellipse/circle. Reversing the ⟨orient⟩ no longer gives the complementary arc, but this complement rotated 180°.




```
\xy 0;/r5pc/:*\dir{*}="p",*+!DR{p};
p+(.5,-.5)*\dir{*}="c",*+!UL{c}**\dir{-}
,"c",{\ellipse<15pt>_,=:a(45){=}}
  ,{\ellipse<>__,=:a(30){-}}
,{\ellipse(1,.4){.}}
  ,{\ellipse(1,.4)_,=:a(120){-}}
,{\ellipse(,.75){.}}
  ,{\ellipse(,.75)_,^,^{-}}\endxy
```

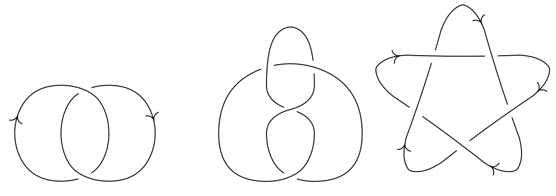
28 Knots and Links feature

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{knot}

This feature provides a language for specifying knots, links and general arrangements of crossing strings.

This knot feature is really a 'construction kit', providing pieces which may be placed appropriately to form knots and links. The types of pieces provided are of two kinds: the "crossings", representing one string crossing over or under another; and "joins" which are used to connect what would otherwise be loose ends. Several types of each are provided, along with a simple way of specifying where to place arrowheads and labels.

All the pieces ultimately use curves from the curve extension, usually indirectly via the arrow feature. As such, processing can be memory-intensive and may seem rather slow. All the warnings and advice given elsewhere on techniques to handle pages and individual diagrams with many curves are especially applicable when using this feature.



simple link   figure-8 knot   cinquefoil

Crossings

A "crossing" is intended to represent two strings passing close by, but not meeting. The macros provided specify typesetting within a square cell of coordinate values; using a non-square basis alters this shape, but see also note 28c below, for the technique that was used in the "cinquefoil" example above.

Notes

28a. Several families of crossing are provided. Those having names \v... and \h... are designed to stack respectively vertically and horizontally. More precisely the current ⟨pos⟩ starts at the top-left corner and finishes at either the bottom-left or top-right. Say that a crossing is either a 'vertical crossing' or 'horizontal crossing' respectively.

This certainly applies to the \..cross.. and \..twist.. families, see figure 20 in which the strings enter and leave the square all with vertical tangents or all with horizontal tangents. Indeed *all* crossings are either vertical or horizontal, with the final letter indicating which for the \xover.. families.

Furthermore there is a natural *orientation* for each crossing, as well as along each strand. This corresponds to the order in which ink is applied to the printed page, following the natural parametrization of each strand as a curved connection or arrow. This orientation determines whether a crossing is 'over' (mathematically, positive or right-handed) or 'under' (mathematically, negative or left-handed). It is used in determining the location of labels and the direction of arrowheads placed along the strings. Note that \..cross.. and \..twist.. crossings may set the same curves, but with different orientation and label-positioning.

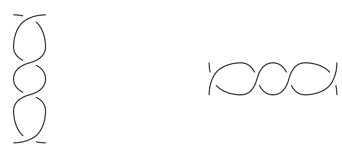
Figure 20 displays the orientation on all the crossings, grouping them into subfamilies consisting of

Syntax		Action
⟨knot-piece⟩	⟶ ⟨piece⟩⟨scale⟩⟨knot-labels⟩	interpret knot-piece
⟨piece⟩	⟶ ⟨crossing⟩ | ⟨join⟩	piece is a crossing[28a] or a join[28l]
⟨scale⟩	⟶ ⟨empty⟩ | `-` | `[`⟨num⟩`]`	invert or scale the knot piece[28b];
	| `~`⟨pos⟩⟨pos⟩⟨pos⟩⟨pos⟩	alter size and shape[28c] using the ⟨pos⟩s
⟨knot-labels⟩	⟶ ⟨empty⟩ | ⟨knot-tips⟩⟨knot-labels⟩	arrowtips at ends, aligned with orientation
	| ⟨where⟩⟨what⟩⟨knot-labels⟩	list[28k] of arrowtips, breaks and labels[28e]
	| `@`⟨adjust⟩⟨knot-labels⟩	adjust hole[28d] position for a ⟨crossing⟩; adjust other parameter[28n] for a ⟨join⟩.
⟨knot-tips⟩	⟶ `==` | `=!`	arrowtips[28k] at both/neither end
	| `=<` | `=>`	arrowtips[28k] also at start/finish
⟨where⟩	⟶ `|` | `|`⟨adjust⟩	'over' string on a ⟨crossing⟩;[28f] middle[28m] place on a ⟨join⟩.
	| `<` | `<`⟨adjust⟩	initial portion of 'under' string on a ⟨crossing⟩;[28f] earlier[28m] place on a ⟨join⟩.
	| `>` | `>`⟨adjust⟩	final portion of 'under' string on a ⟨crossing⟩;[28f] later[28m] place on a ⟨join⟩.
⟨adjust⟩	⟶ `(+`⟨num⟩`)` | `(-`⟨num⟩`)`	adjustment[28k] from current value of parameter
	| `(=`⟨num⟩`)` | `(`⟨num⟩`)`	set parameter value[28k]
⟨what⟩	⟶ `>` | `<`	arrowhead aligned with/against orientation[28i]
	| `\knothole` | `\khole`	leave hole in the string[28j]
	| `{`⟨text⟩`}`	set[28g] ⟨text⟩ as label, using `\labelstyle`
	| `{*`⟨object⟩`}`	drop ⟨object⟩[28h]
	| `{`⟨anchor⟩⟨it⟩`}`	⟨break⟩ or label[28h] as on an ⟨arrow⟩
	| `|`	null-break[28k]

Figure 19: ⟨knot-piece⟩ construction set

right-handed, left-handed and non-crossings. Also indicated are the default positions for labels and arrow-tips; each piece uses the same code for tips and labels, e.g. `\vover<>|>>><{x}|{y}>{z}`.

The `\x...` crossings do not stack easily since their tangents are at 45° to the coordinate axes. It is the last letter in the name which denotes whether the particular crossing is vertical or horizontal. On the other hand `\vover`, `\vunder` etc. stack vertically on top of a `\vcross`, `\vtwist` etc.; similarly `\hover` stacks at the left of `\hcross`, `\htwist` etc.



```
$$\xy 0;/r1pc/:
 ,{\vunder\vtwist\vtwist\vunder-}\endxy
\qquad\qquad\qquad \xy 0;/r1pc/:+(0,-1.5)
 ,{\hover\hcross\hcross\hover-}\endxy$$
```

28b. The above examples also show how to use `-` to get the mirror-image of a particular crossing. Any numerical scale factor can be used by enclosing it within `[..]` e.g. `[2.3]` scaling a single piece without affecting the rest of the picture. The scale-factor *must* occur before any label or arrow-tip specifiers, see below). Vertical crossings remain vertical under scalings; the current ⟨pos⟩ still moves by 1 coordinate unit in the 'down' direction. Similarly horizontal crossings remain horizontal. The single character `-` is a shorthand version for `[-1]`, effectively giving a half-turn rotation in a rectangular basis.

28c. A knot-piece need not be rectangular. By specifying `~`$\langle pos\rangle_1\langle pos_2\rangle\langle pos_3\rangle\langle pos_4\rangle$ the four corners `UL`,

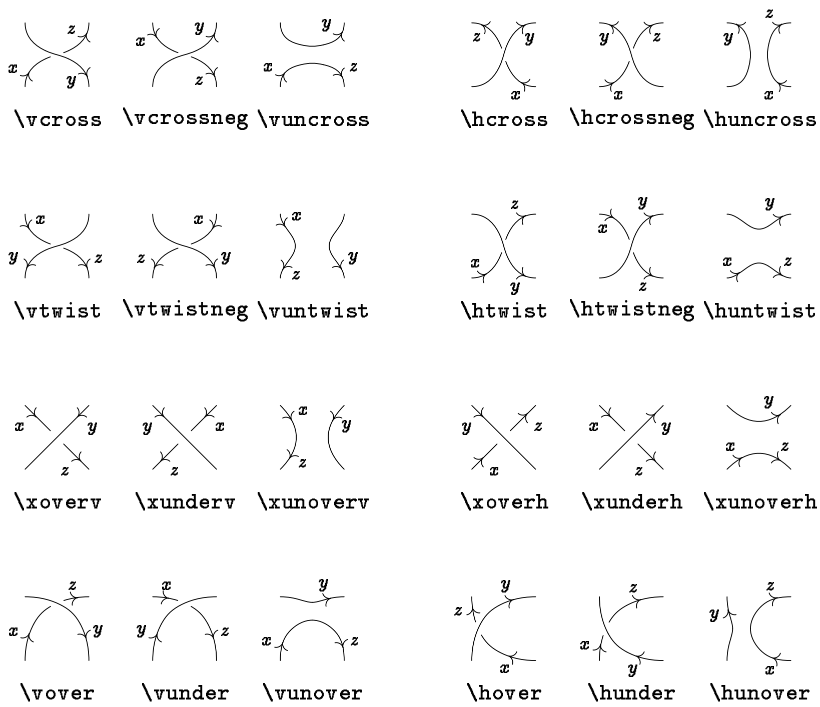


Figure 20: knot crossings with orientations and label positions

UR, DL, DR are set to the given ⟨pos⟩s respectively. The local basis is established so that

$$r\text{–hop} \;\leftrightarrow\; \tfrac{1}{2}\big(\langle\mathrm{pos}_2\rangle - \langle\mathrm{pos}_1\rangle + \langle\mathrm{pos}_4\rangle - \langle\mathrm{pos}_3\rangle\big)$$
$$u\text{–hop} \;\leftrightarrow\; \tfrac{1}{2}\big(\langle\mathrm{pos}_1\rangle - \langle\mathrm{pos}_3\rangle + \langle\mathrm{pos}_2\rangle - \langle\mathrm{pos}_1\rangle\big)\ .$$

28d. With a non-rectangularly shaped piece it will usually be necessary to adjust the place where the 'hole' occurs in the 'under' string. This is done by specifying `@(`⟨num⟩`)`, with $0 \le \langle\text{num}\rangle \le 1$ being the parameter value of the new location for the hole.

28e. The `knot` feature allows for the easy placement of the following objects along the strings of a crossing:

- labels on the strings;
- arrowheads for direction or orientation;
- holes in strings, allowing another string to be drawn passing over.

28f. The characters `<`, `>` and `|` are used to indicate to which string portion the object is associated; with `|` denoting the string which crosses the other, while `<` and `>` denote the initial and final portions of the 'crossed' string.

28g. A simple label enclosed in braces, e.g. `\vcross>{x}` is set in math-mode using `\labelstyle`, at a pre-determined place on the string portion, shifted in either the 'above' or 'below' direction from the curve at this point. (For each crossing depicted in figure 20 only default values are used for the place and shift-direction.)

28h. If the first character within the braces `{..}` is `*` e.g. `\htwist>{*`⟨object⟩`}`, then a general ⟨object⟩ may be placed as a label. Furthermore if the first character is `^` or `_` or `|`, then the interpretation is, e.g. `\vtwist<{^`⟨anchor⟩⟨it⟩`}`, as in 15 to place ⟨it⟩ as a label along an `\ar` of the `arrow` feature.

28i. A second character `<` or `>` specifies that an arrowhead should appear at the pre-determined place on the chosen string. Here `>` denotes an arrowhead pointing with the natural orientation, while `<` points against. Due to the curvature of the strings, it is usually best to `\UseComputerModernTips` rather than normal arrow-tips.

28j. To generate a 'hole' use `\knothole`, or simply `\khole`, as following token. This generates a 'break', in the sense of 21j. Indeed such a 'hole' is used to separate the two portions of the 'crossed' string. Default size for the hole is 5pt, which is alterable via `\knotholesize{`⟨dimen⟩`}`; normally used to set the size for *all* holes in a diagram.

28k. If the resulting `\khole` is either too large or perhaps non-existent, this could be due to a technicality in the way breaks in curves are handled. This problem should not occur with the standard crossings, using a rectangular basis, but it may occur with non-rectangular bases. An easy 'fix' is to include an extra *null-break* on the string, using `<|`, `>|` or `||`, which should place the zero-sized break at parameter value .5 on the curve. The specification should precede a `\khole` at a higher parameter value, or come after one at a lower value.

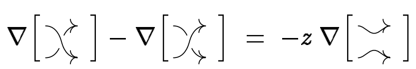
Multiple breaks, arrow-heads and labels may be specified along the two strings of a crossing; simply place their specifications one after another; e.g. `<>|>>><{x}|{y}>{z}` was used in figure 20.

The only proviso is that all 'breaks' along a single strand must occur with increasing order of parameter position. On the 'crossed' string this includes the automatic 'hole' to create space for the other string. Hence it is advisable to use just the `(+..)` and `(-..)` variants for small adjustments, and to keep these correctly ordered.

Adjustment of position along the strings is achieved using a $\langle$factor$\rangle$, as in `\vover|(+.1)>`. Allowed syntax is `(`$\langle$sign$\rangle\langle$num$\rangle$`)` where $\langle$sign$\rangle$ is `+` or `-` to increment or decrement from the predefined value. Also allowable are `=` or $\langle$empty$\rangle$ to set the parameter position to $\langle$num$\rangle$, which must lie between 0 and 1 to have any meaning.

Arrowheads can also be placed at either, or both, ends of of the strings forming a crossing. This governed by a pair of booleans, initially `{FF}`. This is changed for *all* subsequent strings in a diagram by `\knottips{..}` where the recognised values are `{FF}`, `{FT}`, `{TF}` and `{TT}`, denoting tips (`T`) or not (`F`) at the start and end of each string. To add arrowtips at the start of strings in a particular crossing, append the 2-character combination `=<`; similarly `=>` adds tips at the ends, if not already requested. The combinations `==` and `=!` specify both (`{TT}`) and none (`{FF}`) respectively. These 2-character pairs can be mixed in with any specifications for labels and breaks, etc. Multiple pairs compound their effect; in particular `=<=>` gives the same result as `==`, while `=!=<` is needed to change `{FT}` into `{TF}`.

These are best used with single pieces, as in the following equation.



```
\UseComputerModernTips \knottips{FT}
\def\Conway#1{\mathord{\nabla\Bigl[\,
 \raise5pt\xybox{0;/r1pc/:#1}\,\Bigr]}}
$$
\Conway\htwist - \Conway\htwistneg
 \;=\; -z\,\Conway\huntwist $$
```

Joins

28l. The "joins" are used to connect the loose ends of crossing strings. In particular "loops" and "caps" are for placing on the ends of the horizontal or vertical 'twist' and 'cross' crossings. These leave the current $\langle$pos$\rangle$ fixed. The "bends" join non-adjacent crossings of the same type, either horizontal or vertical.

The `\xcap..` pieces are designed to join adjacent `\xover..` pieces; they move the current point either vertically or horizontally, as appropriate. Finally the `\xbend..` pieces allow for smooth joins of 45° slopes to horizontal or vertical slopes. For these the actual positioning of the piece, see figure 21, is not entirely obvious.

Figure 21 displays the orientation on the standard joins. Also indicated are the default positions for labels and arrow-tips; each piece uses the same code for tips and labels, e.g. `\vloop<>|>>><{x}|{y}>{z}`. Furthermore the current $\langle$pos$\rangle$ before the piece is drawn is marked using $^\circ$, while the $\langle$pos$\rangle$ afterwards is indicated by $^\times$ or $^+$.

The ability to scale in size and place arrow-tips, breaks, labels etc. apply also to $\langle$join$\rangle$ pieces. The only difference is...

28m. The three places referred to by `<,|,>` are all on a single string. In particular `|` is always at the middle of the $\langle$join$\rangle$, whereas `<` and `>` are at *earlier* and *later* parameter values respectively. Any adjustments[28k] involving breaks should occur in increasing parameter order.

28n. A parameter can be altered, using `@`$\langle$adjust$\rangle$, to effect subtle adjustments to the shape of any join. Within a rectangular basis the horizontal or vertical tangents are preserved and overall reflection or rotation symmetry is preserved. Thus this parameter affects the 'flatness' of a cap or loop, or the amount of curvature is s-bends and z-bends. For `\xcap..`s and `\xbend..`s the 45° angle is altered; this is especially useful to match the tangents when a knot-piece has been specified using the technique of note 28c.

The normal range for these parameters is between 0 and 1. Other values can be used with interesting results—the parameter determines the location of control points for a Bézier cubic curve.

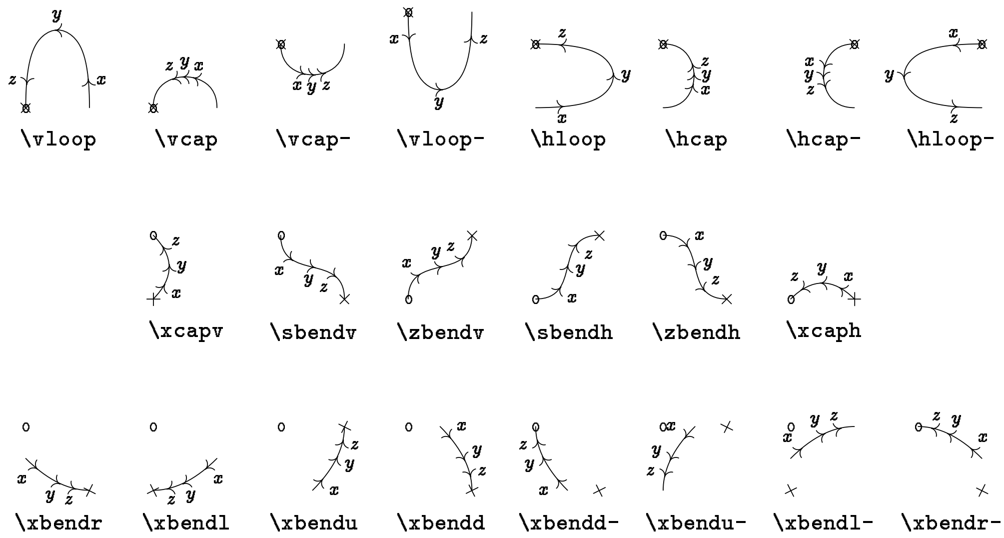
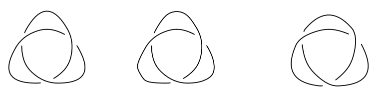


Figure 21: knot joins, with orientations, labels and shifts

piece	value	effect on...
`\..cap`	.25	flatness of cap;
`\..loop`	.75	flatness of loop;
`\sbend..`	.75	curvature in the 's';
`\zbend..`	.75	curvature in the 'z';
`\xcap..`	.5	height of cap, slope at base;
`\xbend..`	.5	curvature, slope at base.

The following example gives three ways of specifying a 'trefoil' knot, using the poly feature to establish the location of the vertices for knot-pieces. In each the ⟨crossing⟩s are calculated to fit together smoothly; a different way of creating ⟨join⟩s is used in each. Also the third displays subtle changes of the [28n]join control.



```
\def\TrefoilA{\xygraph{!{0;/r.75pc/:}
 !P3"a"{~>{}}!P9"b"{~:{(1.3288,0):}~>{}}
 !P3"c"{~:{(2.5,0):}~>{}}
 !{\vover~{"b2"}{"b1"}{"a1"}{"a3"}}
 !{"b4";"b2"**\crv{"c1"}}
 !{\vover~{"b5"}{"b4"}{"a2"}{"a1"}}
 !{"b7";"b5"**\crv{"c2"}}
 !{\vover~{"b8"}{"b7"}{"a3"}{"a2"}}
 !{"b1";"b8"**\crv{"c3"}}}}
%
\def\TrefoilB{\xygraph{!{0;/r.75pc/:}
 !P3"a"{~>{}}!P9"b"{~:{(1.3288,0):}~>{}}
 !P3"c"{~:{(2.5,0):}~>{}}
 !{\vover~{"b2"}{"b1"}{"a1"}{"a3"}}
 !{\vcap~{"c1"}{"c1"}{"b4"}{"b2"}@(+.1)}
 !{\vover~{"b5"}{"b4"}{"a2"}{"a1"}}
 !{\vcap~{"c2"}{"c2"}{"b7"}{"b5"}@(+.2)}
 !{\vover~{"b8"}{"b7"}{"a3"}{"a2"}}
 !{\vcap~{"c3"}{"c3"}{"b1"}{"b8"}}}}
%
\def\TrefoilC{\xygraph{!{0;/r.75pc/:}
 !P3"a"{~>{}}
 !P12"b"{~:{(1.414,0):}~>{}}
 !{\vover~{"b2"}{"b1"}{"a1"}{"a3"}}
 !{\save 0;"b2"-"b5":"b5",
   \xcaph @(+.1)\restore}
 !{\vover~{"b6"}{"b5"}{"a2"}{"a1"}}
 !{\save 0;"b6"-"b9":"b9",
   \xcaph @(+.2)\restore}
 !{\vover~{"b10"}{"b9"}{"a3"}{"a2"}}
 !{\save 0;"b10"-"b1":"b1",
    \xcaph @(+.3)\restore} }}
$$\TrefoilA\quad\TrefoilB
   \quad\TrefoilC$$
```

Changing the string-style

It is not necessary to use solid curves; any style available to curves and arrows can be chosen using...

`\knotstyle{`⟨char⟩`}`	use `\dir{`⟨char⟩`}` for knots
`\knotstyles{`⟨char⟩`}{`⟨char⟩`}`	two styles on ⟨crossing⟩s
`\knotSTYLE{`⟨code⟩`}`	use ⟨code⟩ to set styles

In each case the new style applies to *all* subsequent knot pieces. The latter case allows use of object ⟨modifier⟩s. The ⟨code⟩ consists of two groups `{..}{..}`, each containing ⟨arrow⟩ forms, as in 15 and notes 21m, 21r. Only the first ⟨arrow⟩ form is used with ⟨join⟩s whereas the two forms are used respectively with the two strings of a ⟨crossing⟩ in the order that they are drawn.

Part IV
Drivers

This part describes 'drivers' that customise the parts of the DVI file generated from XY-pictures to exploit special capabilities of particular DVI driver programs through TEXś \special command. This makes the DVI files non-portable but is needed for full support of some of the XY-pic extensions (described in part II).

29 DVIPS driver

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{dvips}

This driver provides support for *all extensions* when using the DVIPS driver by Tomas Rokicki [11]. It has been tested with dvips version 5.55a and dvipsk version 5.58f.
Supported \special effects are...

- colour, using direct color specials and POSTSCRIPT
- crayon colours
- POSTSCRIPT back-end
- variable line-widths and poly-lines, using POSTSCRIPT
- extra frames and fills, using POSTSCRIPT
- patterns and tiles, using POSTSCRIPT
- rotated/scaled diagrams and text, using POSTSCRIPT
- TPIC

30 OzTeX driver

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{oztex}

This driver provides the necessary interface to support the POSTSCRIPT back-end and other POSTSCRIPT effects when using the DVI driver of version 1.8 of OzTEX by Andrew Trevorrow,[12] *Earlier versions of OzTEX should instead use the driver option* \xyoption{17oztex}.

Effects such as colour, line-thickness and rotated or scaled diagrams are only partially supported in that the effects cannot be applied to any text or symbols placed using fonts. This is due to the nature of OzTEX ⟨driver⟩, whose optimization of the placement of font-characters precludes the applicability of such effects. Furthermore the POSTSCRIPT dictionary must be available in a file called global.ps or appended to the OzTeXdict.pro. However with version 1.8 and later of OzTEX, there is the alternative of using the dvips ⟨driver⟩, which does support all the POSTSCRIPT effects available in XY-pic.

Note: To use XY-pic effectively with OzTEX requires changing several memory parameters. In particular a 'Big-TEX' is needed, along with an increase in the pool_size parameter. Explicit instructions are contained in the file INSTALL.OzTeX of the XY-pic distribution.
Supported \special effects are...

- POSTSCRIPT back-end
- variable line-widths and poly-lines, using POSTSCRIPT
- extra frames and fills, using POSTSCRIPT
- patterns and tiles, using POSTSCRIPT
- rotated/scaled diagrams recognized but not supported
- colour, using POSTSCRIPT, but not of font-characters
- crayon colours, restricted as above

31 OzTeX v1.7 driver

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{17oztex}

This option provides the necessary interface to support the POSTSCRIPT back-end and other POSTSCRIPT effects when using the DVI driver of version 1.7 of OzTEX by Andrew Trevorrow,[13] *Later versions of OzTEX should instead use the driver option* \xyoption{oztex}. Upgrading to version 1.8 ofOzTEX is recommended.

Does not support rotations, scaling and coloured text within diagrams and the POSTSCRIPT dictionary must be available in a file called global.ps.

Note: To use XY-pic effectively with OzTEX requires changing several memory parameters. In particular a 'Big-TEX' is needed, along with an increase in the

[12] OzTEX 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⟩.

[13] OzTEX 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⟩.

pool_size parameter. Explicit instructions are contained in the file INSTALL.OzTeX of the XY-pic distribution.
Supported \special effects are...

- POSTSCRIPT back-end
- variable line-widths and poly-lines, using POSTSCRIPT
- extra frames and fills, using POSTSCRIPT
- patterns and tiles, using POSTSCRIPT
- rotated/scaled diagrams recognized but not supported
- colour, using POSTSCRIPT, but not of font-characters
- crayon colours, restricted as above

32 Textures driver

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{textures}

This driver provides support for version 1.7+ of Blue Sky Research's TEXTURES application for Macintosh[14]. It incorporates support for colour and all of XY-pic's POSTSCRIPT effects. Earlier versions of TEXTURES should instead use the driver option \xyoption{16textures}.

Notice that version 1.7 suffers from a printing bug which may cause a POSTSCRIPT error. It is kludged by making sure the first page has been shown in the viewer before any pages with diagrams are sent to the printer.
Supported \special effects are...

- colour, both on-screen and with POSTSCRIPT
- crayon colours
- POSTSCRIPT back-end
- variable line-widths and poly-lines, using POSTSCRIPT
- extra frames and fills, using POSTSCRIPT
- patterns and tiles, using POSTSCRIPT
- rotated/scaled diagrams and text, using POSTSCRIPT

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

33 Textures v1.6 driver

Vers. 3.0 by Ross Moore ⟨ross@mpce.mq.edu.au⟩
Load as: \xyoption{16textures}

This driver provides support for versions 1.5b and 1.6 of Blue Sky Research's TEXTURES application for Macintosh[15]. It, incorporates support for POSTSCRIPT colour and the XY-ps POSTSCRIPT back-end. This will *not* work with versions 1.7 and later; these require the ⟨driver⟩ option \xyoption{textures}.
Supported effects are...

- POSTSCRIPT back-end
- variable line-widths and poly-lines, using POSTSCRIPT
- extra frames and fills, using POSTSCRIPT
- patterns and tiles, using POSTSCRIPT
- rotated/scaled diagrams and text, using POSTSCRIPT
- colour, using POSTSCRIPT
- crayon colours

Appendices

A Answers to all exercises

Answer to exercise 1 (p.6): 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.6): Use the ⟨pos⟩ition <X,Y>+"ob".

Answer to exercise 3 (p.6): 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.*,

$$\underbrace{\text{The }\ldots}_{w}\underbrace{\qquad\quad}_{w}\times$$

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

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

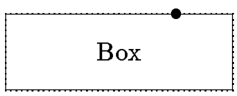
```
%
% set up and mark A, B, C, and D:
(0,0)="A"  *\cir<1pt>{}*+!DR{A},
(7,10)="B" *\cir<1pt>{}*+!DR{B},
(13,8)="C" *\cir<1pt>{}*+!DL{C},
(15,4)="D" *\cir<1pt>{}*+!DL{D},
%
% goto intersection and name+circle it:
{"A";"B":"C";"D",x} ="I" *\cir<3pt>{},
%
% make dotted lines:
"I";"A"**{} +/1pc/;-/1pc/ **\dir{..},
"I";"D"**{} +/1pc/;-/1pc/ **\dir{..}
%
\endxy
```

A `?!...` ⟨place⟩ could also have been used.

Answer to exercise 5 (p.9): To copy the p value to c, *i.e.*, equivalent to "p".

Answer to exercise 6 (p.9): 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.9): The code typesets the picture



Answer to exercise 8 (p.10): This does the job, saving each point to make the previous point available for the next piece:

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

Notice how we close the line by first saving `s0`, the last point visited, such that the first point will be connected to it.

Answer to exercise 9 (p.10): 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 10 (p.11): 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: a and a+.

Answer to exercise 11 (p.11): 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 12 (p.13): 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 13 (p.13): 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.17): 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.19): 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.19): 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.19): 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.23): Here is how:

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

Answer to exercise 20 (p.23): The `*\cir {}` operation changes c to be round whereas `*\frm {o}` does not change c at all.

Answer to exercise 21 (p.23): 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.27): 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.36): 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.37): The author did

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

Answer to exercise 25 (p.37): 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.37): The author used

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

Answer to exercise 27 (p.38): The author used

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

Answer to exercise 28 (p.39): The author used

```
\xy *{\circ}="b" \ar@(ur,ul) c
 \ar@{.>}@(dr,ul) (20,0)*{\bullet}
\endxy
```

Note that it is essential that the curving specification comes after the arrow style.

Answer to exercise 29 (p.41): Here is the code used to typeset the *pasting diagram* in figure 16.

```
\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.43): 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.46): 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.46): The author did

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

Answer to exercise 33 (p.46): Here is how:

```
\objectheight{1pc} \objectwidth{3pc}
\xymatrixrowsep={0pc}
\everyentry={\drop\frm{-}}
\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.47): The first A was named to allow reference from the last:

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

Answer to exercise 35 (p.49): 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
```

B Version 2 Compatibility

Vers. 3.0 by Kristoffer H. Rose ⟨kris@diku.dk⟩
Load as: \xyoption{v2}

This appendix describes the special 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 loaded. The compatibility is available either as an XY-option or through the special files xypic.sty and xypic.tex described below.

There are a few exceptions to the compatibility: the features described in §B.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 with old diagrams.

The remaining sections list all the obsolete commands and suggest ways to achieve the same things using XY-pic 3.0+, *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![16]

B.1 Unsupported incompatibilities

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

- Automatic 'shortening' of arrow tails by |<< breaks 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. The version 2.6 construction t_0;t_1*(x,y) should be replaced by the rather long but equivalent construction

  {t_0;p+/r/:t_1="1";p+/u/,x;(0,0);:
  "1";p+/r/,y;(0,0);::(x,y)}

  In most cases, however, the construction t_0;t_1**{}?(x), possibly with a trailing +/^.../, suffices instead.

- Using t_0;t_1:(x,y) as the target of an arrow command does not work. Enclose it in braces, *i.e.*, write

  {t_0;t_1:(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.

[16] In addition a few of the experimental facilities supported in v2.7–2.12 are also supported.

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 3 is slightly different from that of version 2.6 which had some spacing bugs.

B.2 Obsolete kernel features

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 $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX idioms

Some idioms from $\mathcal{A}\mathcal{M}\mathcal{S}$-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 $\mathcal{A}\mathcal{M}\mathcal{S}$-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 $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX or $\mathcal{A}\mathcal{M}\mathcal{S}$-LATEX or whatever) if possible. Please use the \txt object construction described in §6.3 directly since it is more general and much more efficient!

Obsolete state

Upto version 2.6 users could access the state variables \cL, \cR, \cH, and \cD, which are defined.

From v2.7 to 2.12 users could use the names of the state \dimen registers \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. Now the same effect can be achieved using ⟨corner⟩s but v2 defines the aliases.

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:

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

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 3.0+ (it was never accurate in version 2).

B.3 Obsolete extensions & features

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-}`
`\circled`	`\drop\frm{o}`
`\Circled`	`\drop\frm{oo}`

Matrices

The `\diagram` ⟨rows⟩ `\enddiagram` command is provided as an alias for `\xymatrix{` ⟨rows⟩ `}` 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.

B.4 Obsolete loading

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 3.0) just loads the v2 feature.

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

B.5 Compiling v2-diagrams

In order to make it possible to use the new compilation features even on documents written with XY-pic v2, the following command was added in v2.12:

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

which is like the ordinary diagram command except the result is compiled (see note 5e). 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⟩ `}`
`\NoCompileAllDiagrams`
`\ReCompileAllDiagrams`

will apply `\xycompileto{`⟨prefix⟩n`}{`…`}` to each diagram with n a sequence number starting from 1. Use `\CompileMatrices` and `\CompilePrefix` instead!

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}`).

C Common Errors

In this appendix we describe some common cases where small mistakes in XY-pictures result in TEX error messages that may seem cryptic.

! Box expected.

! A ⟨box⟩ was supposed to be here. This message is common when an XY-pic ⟨object⟩ is mistyped because then XY-pic expects a TEX ⟨box⟩ construction.

! LaTeX Error: Bad math environment delimiter. This error happens while reading an incomplete compiled picture (such a beast is created when XY-pic crashes during compilation due to a syntax error or other such problem).

To Do: Also include the more obscure ones...

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*, second edition, 1990.

[2] American Mathematical Society. *AMS-LATEX Version 1.1 User's Guide*, 1.1 edition, 1991.

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

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

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

[6] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

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

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

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

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

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

[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*. DIKU, University of Copenhagen, Universitetsparken 1, DK–2100 København Ø, 3.0 edition, June 1995.

[15] Kristoffer H. Rose and Ross R. Moore. XY-pic complete sources with TEXnical commentary. not yet available, June 1995.

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

Index