How to Generate LaTeX Picture Environments Using the GaPFilL Method

Herbert Möller*

ABSTRACT. Drawing programs or geometry software and Perl filter programs are used to create conveniently even complicated figures with the LaTeX `picture` environment. The filter programs parse PostScript files and generate LaTeX code ready for use. The new method will be explained with two filter programs for the geometry software Cabri-géomètre™ II. The first program only requires the package `ebezier` [2]. Therefore the output is driver independent. The second filter in addition supplies the new package `pict2e` [4].

1. INTRODUCTION

The package `pict2e`, created by HUBERT GÄSSLEIN and ROLF NIEPRASCHK, eliminates the most serious restrictions of the LaTeX `picture` environment. Henceforth, essentially the problem of positioning the objects remains. An important suggestion, using the package PICTEX of MICHAEL J. WICHURA [9], was introduced in 1999 by ROBERT W.D. NICKALLS with the MS DOS filter program 'MathsPIC' [7]. Afterwards, it was extended by him and by APOSTOLOS SYROPOULOS to a Perl filter program [8]. However, the convenient input achieved is on the expence of about 90 further commands.

Hence the question arose whether LaTeX figures can also be constructed without using control commands. This seemed to be feasible because the PostScript files generated by graphic programs and used by the LaTeX command `\special` evidently contain all necessary information. The answer is a method which consists of four steps:

(1) Generating the desired figure with a drawing program or with geometry software;
(2) "Printing" (or exporting) the construction as a PostScript file;
(3) Applying a filter program to the PostScript text;
(4) Copying the resulting code or parts of it into a LaTeX document.

*© Copyright 2006 by H. Möller (`mollerh@math.uni-muenster.de`). This documentation may be distributed and/or modified under the conditions of the LaTeX Project Public License.

Because of this structure, the method is called “GaPFilL” (*Graphics as PostScript Filtered for LaTeX*). Since the steps (1) and (2) and the packages used in the LaTeX document may vary, it is necessary to apply different filter programs. In the following the context will be described as an example by means of two prototypes: `CABebez.pl` only requires the macro package `ebezier`; `CABpict.pl` in addition uses the above mentioned package `pict2e`.

2. The Drawing Program or Geometry Software

Most programs which generate graphical objects are suited for the GaPFilL method. To create diagrams and simple illustrations, it is sufficient to use drawing programs belonging to office packages. The fine drawing program `Draw` of `OpenOffice.org` is even free of charge. For the representation of geometrical facts, geometry software is preferable since, apart from the construction of the objects, it offers a lot of transformations and combinations such as reflection, bisection of angles, transfer of measurement and generation of loci.

In the educational system, Cabri-géomètre™ II (in the following: Cabri Geometry) is a powerful “dynamic” geometry software with wide distribution for the operating systems Windows® and Mac®OS. Since 1987 it was developed by Jean-Marie Laborde and Franck Bellemain at the ‘Institut d’Informatique et Mathématiques Appliquées’ of the Joseph Fourier University in Grenoble. Among other things, due to numerous courses of further education for teachers and due to the implementation of adapted versions in the hand-held computers TI-92, Voyage™ 200 and other calculators of Texas Instruments® it is widely used.

Since many people are familiar with these programs, we will only describe the pecularities which are important for the generation of LaTeX figures. In 2004, improved versions Cabri-géomètre™ II plus both for Windows and Macintosh computers were published. The differences will be mentioned where appropriate.

3. The Use of Colours

With the aid of colours additional information is conveyed to the filter program. Colouring of objects in the LaTeX `picture` environment objects is carried out on demand with commands of the `color` package when the work is to be finished

(see Section 8). The names of the colours used in the following come from the Macintosh version of Cabri Geometry, where only eleven colours are available. The Mac OS version of Cabri Geometry II plus has a palette with 36 colours. In both new versions the RGB values of colours may be set by the user. In Section 7 it will be described how to change the assignment of colours in the filter program.

- All *straight lines, rays, line segments, circles and conic sections* which are coloured yellow serve as *drawing aids* because they are ignored by the filter program. The red points shown by Cabri Geometry will be disregarded by the filter program too. Straight lines, rays, parabolas and hyperbolas are cut off at the boundary of the drawing section. Since the filter program calculates the exact “bounding box”, these figures should only be used as drawing aid if the resulting shape is not the desired one.
- *Continuous lines and polygons* have to be green.
- *Arrows* must be drawn in violet.
- *Dotted objects (line segments, arrow lines, polygons, circles, arcs)* are obtained by using the colour dark green.
- *Quadratic* and *cubic Bézier curves* are entered as blue polygons with three respectively four corners (see Figures 1 and 2). The corresponding curves may be viewed with Cabri Geometry by using macros. This will be explained in Section 4.

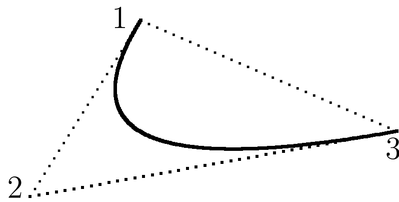

Figure 1

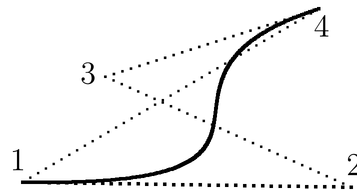

Figure 2

- *Polygons* have to be *closed* by clicking on the starting point or by double clicking. Using the colour navy blue, the last line of the polygon will be suppressed so that *open polygons* can also be entered conveniently.
- *Text or formula places* are positioned by using blue polygons with two corners. For the bounding box to be calculated correctly, the marking line should approximately be the diagonal beginning at the lower left corner of the smallest rectangle enclosing the text or formula. In the LaTeX output a serial number will be written at the position of the starting point.

- The remaining five colours are used for the *filling, hatching and dotting* of areas bordered by *polygons.* For that the filter program cuts up the polygon area into triangles which have the starting point of the polygon as a common corner. To avoid overlap, the given area must be devided by polygons such that for each polygon the triangles which arise from connecting the starting point with the other corners have at most one side in common with the other triangles (see Figures 3 and 4).

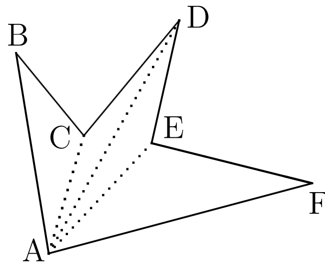

Figure 3

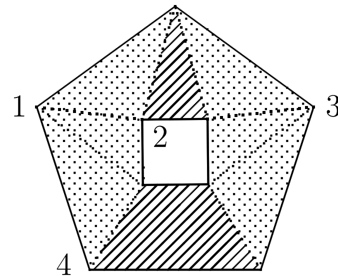

Figure 4

Red *polygons* are *filled* without boundery. For the *hatching with visible boundary lines* the colour orange has to be chosen for the polygon; *without boundary lines* the colour brown will do it. If the polygon is coloured purple, the area will be *dotted with visible boundary lines*; with the colour violet an area can be *dotted without boundary lines.* The previous assignments of colours are summarized in the following table.

Type	Colour
auxiliary line	yellow
unbroken object	green
arrow	violet
dotted object	dark green
Bézier curve or text	blue
open polygon	navy blue
filling without border	red
hatching with border	orange
hatching without border	brown
dotting with border	purple
dotting without border	dark brown

- With Cabri Geometry semicircles and quadrants of a circle, which in LaTeX are also connected with the `\oval` command, must be constructed as circular arcs determined by three points on a circle. For those circular arcs which have

at their ends radii parallel to the coordinate axes, the input is simplified by using coloured circles. The next table contains the assigned colours. With the Macintosh version of Cabri Geometry, the respective colours of the objects can be preset in the "standard settings".

Type	Colour
auxiliary circle	yellow
unbroken circle	navy blue
dotted arc or circle	dark green
left semicircle	purple
right semicircle	red
bottom semicircle	orange
top semicircle	dark brown
left bottom quadrant of a circle	blue
left top quadrant of a circle	green
right bottom quadrant of a circle	brown
right top quadrant of a circle	violet

4. Macros for Bézier Curves with Cabri Geometry

In contrast to many drawing programs, Cabri Geometry does not offer the tool 'Bézier curve' which yields a substantial part of the efficiency of LaTeX figures. This deficiency can be compensated by "loci" which are generated with the aid of macros described in the following.

In the mathematical representation we use vectors instead of the spanning points because the geometrical meaning is better known from vector geometry than from complex numbers.

If t with $0 \leq t \leq 1$ is the running parameter, then the *quadratic Bézier curve* spanned by $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3$ can be written in the following form with the abbreviation $t_1 := 1 - t$:

$$\vec{x}(t) = t_1\,(t_1\vec{x}_1 + t\,\vec{x}_2) + t\,(t_1\vec{x}_2 + t\,\vec{x}_3)\,.$$

Since all three linear combinations belonging to the plus signs have the same coefficients t_1 and t, each point of the quadratic Bézier curve can be obtained by dividing three line segments with the same division ratio $t : t_1$. Since

$$t_1\vec{x}_i + t\,\vec{x}_{i+1} = \vec{x}_i + t\,(\vec{x}_{i+1} - \vec{x}_i)\,,\ i = 1,\,2,$$

at first the two "connecting line segments" from $\vec{x}_1$ to $\vec{x}_2$ and from $\vec{x}_2$ to $\vec{x}_3$ are

divided with ratio $t : t_1$. If $\vec{x}_i' := t_1\vec{x}_i + t\,\vec{x}_{i+1},\ i = 1, 2$, are the accompanying "division vectors", then the division of the connecting line segment from $\vec{x}_1'$ to $\vec{x}_2'$ with ratio $t : t_1$ yields the vector $\vec{x}(t)$ belonging to the parameter t.

Correspondingly, the quadratic Bézier curve can be constructed as locus. After the input of three different points P_1, P_2, P_3, the line segment connecting P_1 and P_3 is drawn as "track" of the locus, and a point T is placed on it (see Figure 5).

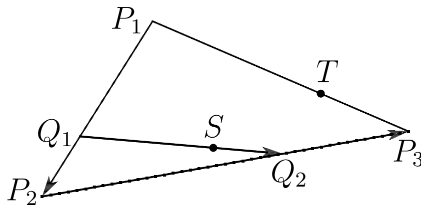

Figure 5

The distances of P_1 and T and of P_1 and P_3 determined with the tool "distance and length" must be transferred into the window of the tool "calculator". They have to be divided. After putting the value of the ratio into the drawing window, it has to be multiplied with the distances of P_1 and P_2 and of P_2 and P_3. To get the right division points Q_1 and Q_2, these resulting lengths have to be marked off on the vectors from P_1 to P_2 and from P_2 to P_3 with the tool "transfer of measurement". In the same way, the point S of the quadratic Bézier curve is constructed as division point on the vector from Q_1 to Q_2.

Calling the tool "locus" and clicking on the points S and T (in this order) the preset number of points of the quadratic Bézier curve will be drawn. Then, in the tool box "macros" the three points P_1, P_2, P_3 and the locus have to be chosen respectively as source objects and as target object by clicking. Finally, with the tool "macro name" the macro can be saved.

The *cubic Bézier curve* determined by the vectors $\vec{y}_1$, $\vec{y}_2$, $\vec{y}_3$, $\vec{y}_4$ is obtained quite similar by six divisions of line segments with the same ratio $t : t_1$ because the abbreviations

$$\vec{y}_j' := t_1\vec{y}_j + t\,\vec{y}_{j+1},\ j = 1, 2, 3, \text{ and } \vec{y}_k'' := t_1\vec{y}_k' + t\,\vec{x}_{k+1}',\ k = 1, 2,$$

enable the representation

$$\vec{y}(t) = t_1\vec{y}_1'' + t\,\vec{y}_2''.$$

Therefore, the cubic Bézier curve can be constructed as locus in a way analogous to that one of the quadratic Bézier curve. At first, four different points have to be entered. Then the connecting line segment of the first and the last point is drawn as track of the locus. A point T which has to be put on the track determines the

division ratio of all six vectors occurring later. The first three vectors connect each of two points entered successively. The next two vectors lie between the three division points. The corresponding division of the sixth connecting vector yields the locus point belonging to T.

The source objects of the macro are the four starting points, the target object is the locus. The Macintosh version of the two macros can be found in the Section "Programs" of the author's web site called *Mathcompass* [6].

5. PostScript Files

The inclusion of graphics with the LaTeX command `\special` is extremely device dependent. This deficiency is reduced considerably by suitable TeX macros which support many important drivers processing PostScript code. The LaTeX packages `graphicx, color` and `pstricks` are examples which, among other possibilities, are described in [5].

Compared with the programming of such TeX macros, the writing of filter programs for the extraction of data from EPS files (EPS = Encapsulated PostScript) relevant for LaTeX requires only little knowledge of PostScript, for example from [1]. For the application with a fitting filter program, a standard EPS printer driver can be used without understanding PostScript. Suitable drivers can be downloaded free of charge from the driver web site of the Adobe Company (`http://www.adobe.com/support/downloads/main.html`).

The following explanations are given for people who want to change or rewrite a filter program. PostScript files can be opened and edited with editor programs like *Alpha* or *BBEdit* with MacOS, *WinEdt* with Windows or *Emacs* with Unix. Frequently these files have more than 1000 lines. But even in this case, only a very small part lying between 'EndProlog' and 'EOF' contains LaTeX relevant data.

For the application of filter programs it is advantageous that all lines containing required numerical values are unambiguously marked at the line end. However, for the individual PostScript generators the marking may be quite different. The lines with the colour data end, among other things, with 'setrgbcolor' or with ':F4'. The object data follow one to three lines later. Here too, the marking may be "concrete" or "abstract". In case of a line segment, it is, for example,

'lineto stroke' respectively '@b'. In the second case, the concrete assignment can be found in the 'Prolog' section.

The meaning and the order of the numerical values from different PostScript generators are not uniform either. For example, a circle can be determined by its centre and radius or by two diagonal corners of its bounding box. Sometimes the x- and y-coordinates of points are exchanged. Furthermore, the numerical values do not always belong to the same unit. Usually it is 'pt' which fits the `picture` environment of LaTeX. There may also occur integer values which have to be multiplied with 0.25 in order to receive the point size.

All these differences can be clarified through systematic trials. At first, each of the different objects has to be constructed with the graphic program. After noting the measurements, the drawing must be exported as a PostScript file. Most of the graphic programs indicate coordinates and lengths in millimetres. The conversion factor to pt is 2.845. Through choice of suitable lattice points or through variation of the figure, the position of the data in question can be determined. If it turns out that the y-coordinates in the PostScript file are opposed to those in the graphic program, either the drawing has to be reflected vertically before saving, or in the filter program, each y-coordinate must be multiplied by -1.

6. Perl

The interpreter language Perl (as abbreviation of "**P**ractical **E**xtraction and **R**eport **L**anguage") was developed around 1988 by Larry Wall at first for scanning arbitrary text files in order to extract and process information. It may be considered as a younger brother of the TeX system published around 1978 by Donald E. Knuth because many persons are improving the systems since the beginning and because both systems with numerous modules and aids are available free of charge in the "Comprehensive TeX Archive Network" (CTAN) and in the "Comprehensive Perl Archive Network" (`http://www.Perl.com/CPAN`) respectively. Meanwhile Perl has become a universal script language with which many kinds of recurring tasks can be automated, for example in the system management of computers and of their peripherals. Along with the great power of Perl since Version 5, it is specially advantageous for the application together

with LaTeX that Perl systems are available on all platforms with wide distribution. Therefore it is not surprising that the CTAN already contains numerous Perl programs.

Whoever has learned a higher programming language (like C, for example) will hardly have difficulties writing or changing Perl filter programs with the aid of some of the online manuals and looking at examples like the ones described in the following section. As with LaTeX, it may be expected that, now having reached the Perl version number 5.8.7, there won't be considerable changes with respect to text filtering. Therefore it is reasonable to provide and to maintain Perl filter programs at this time also for further combinations of graphic programs, PostScript versions and LaTeX packages. That can be achieved in the interest of the LaTeX community by a few persons because the main work has been accomplished developing the filter structure and the subroutines.

7. The filter programs `CABpict.pl` and `CABebez.pl`

The Perl program `CABpict.pl`, which is 32 kB large, has 964 lines including the comment lines beginning with '#' and the lines with closing braces '}'. The program `CABebez.pl` is only 29 lines shorter. Therefore in the following only those lines and blocks are commented which are important for better understanding or which may play a part in modifications.

In case of matching, the lines from `CABpict.pl` are taken. The line numbers, which in both cases don't belong to the programs, may be different even for matching lines. In this documentation, the lines of `CABebez.pl` are marked by a colon behind the numbers.

```
1 #!perl -w
2 # CABpict.pl
3 # (c) Copyright 2006 H. Möller (mollerh@math.uni-muenster.de).
4 # Version 1.1 for Cabri-géomètre II with MacOS 9.x, ... .
5 # This program may be distributed and/or modified under the
  conditions of the LaTeX Project Public License, ... .
...
9 use POSIX('ceil','floor');
```

The first line doesn't represent a comment line because it begins with '#!'. This line may be missing in some Perl versions, for example, if Perl programs are called

by command line input or if no options are used. The option '-w' causes Perl to print error messages. It can also be necessary to place the complete path name before 'perl', for example '/usr/bin/perl'.

Lines 2 to 8 are destined for version references. Line 9 provides two procedures from the Perl modul 'POSIX'. They serve for rounding and can be replaced (with caution) by own (sub-) routines using the command 'int'.

```
11 # Definable by the user:
12 # Unitlength in pt:
13 $ul = 1.0;
14 # Fill factor (for filling with magnification up to 500 %)
15 $fillf = 5;
16 # Point factor:
17 $pointf = 0.3;
18 # Flag for dotting parabolic arcs (1: Dotting)
19 $Qbezflag = 0;
```

```
16: # Bézier factor:
17: $bezf = 2.0;
```

Each of these five parameters may be changed before applying the respective program. Afterwards all parameters should get their original values unless the program is stored under a different name.

Varying the 'unitlength' `$ul` and using a saved PostScript representation, each drawing can be reduced or magnified in the LaTeX `picture` environment. The value `$ul=1.0` yields the figure in its original size from Cabri Geometry.

The 'fill factor' `$fillf` settles the distance of the horizontal line segments filling polygons. With `$fillf=5` figures appear completely filled with screen magnification up to 500 % .

With `pict2e` alone, dotted curves can't be represented because `pict2e` ignores the number of points of Bézier objects. Therefore in `CABpict.pl` the plot commands `\Lbezier` and `\Qbezier` from the package `ebezier` are used additionally. With that the 'point factor'`$ pointf= 0.3` yields the normal point distance (like in Figures 1 to 3). To dot a curve, setting the flag `$Qbezflag=1` one can approximate with arcs of parabolas for which possibly in the LaTeX `picture` environment the number of points at `\Qbezier` is to be adjusted. In the normal case `$Qbezflag=0`, the parabolic arc is drawn with the plot command `\qbezier` from `pict2e`.

In `CABebez.pl` the 'Bézier factor' `$bezf= 2.0` yields a point distance which with normal magnification and in print lets the corresponding line appear closed. Since the memory need of Bézier curves with `ebezier` is often eight times as large as with `pict2e`, `$bezf` may be reduced in case of tight memory.

```
21 # Constants:
22 # Colour names:
23 $yellow = "0.9843900.9511410.020249";
24 $orange = "1.0000000.3927370.009949";
25 $red = "0.8649270.0342110.025910";
26 $purple = "0.9486080.0325630.519234";
27 $violet = "0.2769050.0000000.645487";
28 $navy = "0.0000000.0000000.828138";
29 $blue = "0.0088040.6692610.917967";
30 $green = "0.1215990.7170980.078874";
31 $darkgreen = "0.0000000.3933010.069093";
32 $darkbrown = "0.3359430.1742730.020081";
33 $brown = "0.5657890.4428780.227359";
```

Cabri Geometry writes the three RGB values into the PostScript file with six decimal places and one digit before the decimal point. Through Lines 45 to 47, `CABpict.pl` looks for these lines ending with `␣setrgbcolor␣` and removes the spaces as well as the last word `setrgbcolor`. In this way the corresponding line yields a single "word" which is placed before the respective character strings later. Through the assignment of easily remembered colour names to this "digit words" it is easy to use other or further colour names. The names of 68 (DVIPS-) colours with CMYK values can be found in the header file '`color.pro`'. Without colour values they also appear in the files '`colordvi.sty`' and '`colordvi.tex`' (see [3]).

```
35 # Further Constants:
36 # Pi:
37 $Pi = "3.14159265358979";
38 # Constants in dotted figures:
39 $uli = sp(4 / $ul);
40 $ule = sp(0.8 / $ul);
```

```
38: # Constant in cubic Bézier curves for quarters of a circle:
39: $l90 = "0.552284749830794";
```

To be able to dot areas which are contained in arbitrary closed polygons, the points get absolute coordinates (x, y) with regard to the respective `picture` environment, where x and y are even integers and $x + y$ is divisible by 4. Then

`$uli` and `$ule` deliver the horizontal distance and the length of the 0.8 pt thick line segments which form a point.

Like the "circular number" `$Pi`, `$l90` also represents a number constant which is required for the approximation of circular arcs through cubic Bézier curves. It is deduced in [2] (Page 6).

```
42 @lines = <>;
43 do {
44   $_ = $lines[$i++];
45   if (/ setrgbcolor \s/o) {
46     s/ //go;
47     s/setrgbcolor\s/ /o;
48     $c = $_;
49     $_ = $lines[$i++];
50     s/ moveto//o;
51     s/lineto stroke/stroke/o;
52     s/curveto stroke/curveto/o;
53     s/ setlinewidth stroke//o;
54     s/ lineto//go;
55     if (/stroke/o) {
56       $line[++$#line] = $c.$_;
57     }
58     elsif (/closepath fill/o) {
59       $vector[++$#vector] = $c.$_;
60     }
61     elsif (/arc /o) {
62       $circle[++$#circle] = $c.$_;
63     }
64     elsif (/arcn/o) {
65       $arc[++$#arc] = $c.$_;
66     }
67     elsif (/curveto/o) {
68       do {
69         $conic[++$#conic] = $c.$_;
70         $_ = $lines[$i++];
71         s/ moveto//o;
72         s/curveto stroke/curveto/o;
73       }
74       until $_ !~ /curveto/o;
75     }
76   }
77 }
78 until $i == $#lines;
```

With Lines 42 to 78, the required data are extracted from the PostScript file. Initially the array `@lines` contains all lines. The successive assignment of each individual line to the general "last result variable" `$_` takes place with Line 44. Next, the "colour lines" are looked for and condensed as described above. With Line 48, the result is assigned to the variable `$c`.

In the present version, only the next line has to be analysed subsequently. The commands for searching and replacing in Lines 50 to 54 care for unambiguity and for pure number sequences (apart from the last word). With PostScript versions which contain the data in a later line, it is helpful to attach the intermediate lines and the data line by removing the line breaks.

Depending on the different line endings, the data are collected in five arrays. For example, Line 56 means that the array `@line`, whose last element has the index `$#line`, is extended by one field which holds the preceding colour variable `$c` and the character string of the coordinates of the starting point and the end point of a line segment.

For any arrow, Cabri Geometry writes the coordinates of the four arrowhead corners of a PSTricks style arrow into the PostScript file. With `CABpict.pl` these data are not needed if the colour of the arrow is violet because the plot command `\vector` is then available. Otherwise the arrowhead consists of two filled triangles, and the arrow line is drawn as an unbroken or dotted line segment according to the colouring.

The data of circles can be found in the lines ending with `arc␣`. Additionally a full circle is characterized by 0 and 360 as fourth and fifth value respectively. Therefore, circular arcs entered clockwise in Cabri Geometry can also be recognized in lines ending with `arc␣` because they contain the starting angle and the ending angle instead of 0 and 360. Lines with the data of circular arcs entered anticlockwise end with `arcn`.

The Macintosh version of Cabri Geometry offers the possibility of drawing conic sections determined by five points. In the linked PostScript file, the data of approximating cubic Bézier curves are stored, namely nine for ellipses and four for each open branch. The values from the corresponding lines ending with `curveto stroke` are stored in the array `@conic`. These data can be processed directly with the plot command `\cbezier`.

```
 80 $pflag = 1;
 81 $sflag = 1;
 82 $thicknessflag = 1;
 83 $coun = 0;
 84 $xtex = "";
 85 $mtex = "";

86: $bmax = 500;
89: $btex = "\\documentclass{article}\n\\usepackage{ebezier}\n\n";
```

These initialisations begin with flags for the starting corner of polygons, for the starting values of the bounding box and for the currently selected thickness of line segments. The parameter `$coun` yields the numbers for the places of text and formulas. With `$xtex`, the character string is set up which finally is written as LaTeX text into the Perl output window through a `print` command. Since, in any case, the text markers have to be edited finally, temporary storing in `$mtex` enables the positioning at the end of the LaTeX text where they can easily be found.

If Bézier curves with more than 500 points are generated with the package `ebezier`, it is necessary to enlarge the LaTeX value 500 of `\qbeziermax`. Therefore, with the aid of the variable `$bmax` the maximum number of points of Bézier curves is determined. If `$bmax` exceeds 500, the value of `\qbeziermax` is adjusted with `\renewcommand` in the character string `$btex` which contains the beginning of the corresponding LaTeX program. The insertion of `$btex` and `$mtex` in `$xtex` is described on page 20.

```
 87 # Lines and polygons
 88 $cflag = 1;
 89 foreach (@line) {
 90   @coo = split;
 91   $co0 = $coo[0];
 92   $co2 = (-1) * $coo[2];
 93   $co4 = (-1) * $coo[4];
 94   if (($co0 ne $violet) and ($co0 ne $yellow)) {
 95     if ($cflag) {
 96       $xtex .= "%Lines, arrows, polygons and  Bézier curves\n";
 97       $cflag = 0;
 98     }
 99     if ($co0 ne $blue) {
100       bound($coo[1],$co2);
101       bound($coo[3],$co4);
102     }
```

```
103     if (($co0 ne $red) and ($co0 ne $blue) and ($co0 ne $brown)
104       and ($co0 ne $darkbrown) and ($co0 ne $navy)) {
105       lin($co0,$coo[1],$co2,$coo[3],$co4);
```

Here begins the processing of the data of the arrays. Since these longer parts have a similar structure for most combinations of graphic programs, PostScript versions and LaTeX packages, only essential or typical sections are explained as follows.

The flag `$cflag` ensures that in the LaTeX program, the commentaries pointing out each of the emerging objects don't repeat permanently. With the `split` command the character strings delivered by `foreach` are transformed into lists of character strings which in our case are the numbers and the last words of the evaluated PostScript lines. In the shortened form of `split` used here, the spaces cause the separation of the character string of the last result value `$_`.

Since most branchings depend on at least one colour, first the list element with index 0 which contains the colour, is abbreviated. Subsequently the "vertical reflection" which is necessary in this version takes place multiplying each y-coordinate by -1.

Line 96 yields a typical LaTeX comment line because the character string which is added to `$xtex` by `.=` begins with the `%` sign and ends with the line feed command `\n` of Perl .

With each of the Lines 100 and 101, the subroutine '`bound`' for the determination of the bounding box coordinates is called. It is defined in the Lines 944 to 957. In this case the coordinates of the starting point and the end point of line segments are evaluated. Objects with colour '`blue`' are excluded because the diagonal marking text or formulas is a two point polygon, and because for Bézier curves 51 points are considered.

There are nine subroutines which can be found from Line 390 on (356 with `CABebez.pl`). The order of the subroutines plays no role. Therefore most of these procedures are described at their first appearance.

The subroutine '`lin`' generates all line segments which have to be drawn. In `CABpict.pl`, the extended `\line` command from `pict2e` is available for unbroken line segments, whereas dotted line segments have to be drawn with the command `\Lbezier` from the `ebezier` package. In `CABebez.pl` also all line segments which

don't fulfill the LaTeX conditions for slopes or lengths must be built up with the aid of `\Lbezier`.

```
107     if (($co0 ne $green) and ($co0 ne $darkgreen)) {
108       if ($pflag) {
109         $cb1 = $coo[1];
110         $cb2 = $co2;
111         $pol = $co0." ".$cb1." ".$cb2;
112         $pflag = 0;
113       }
114       else {
115         $pol .= " ".$coo[1]." ".$co2;
116         if (abs($coo[3] - $cb1) + abs($co4 - $cb2) < 2.0E-6) {
117           $poly[++$#poly] = $pol;
118           $pflag = 1;
```

Here, for each polygon which has to be filled, dotted or hatched, the array `@poly` emerges which contains the coordinates of the corners.

```
123   if ($co0 eq $violet) {
124       $xtex .= "%Arrow\n";
...
127       $dx = $coo[3] - $coo[1];
128       $dy = $co4 - $co2;
129       $len = sp(abs($dx));
130       if ($len > 1.0E-3) {
131         @p = best(abs($dy / $dx));
132         $psx = sp($p[1]) * ($dx <=> 0);
133         $psy = sp($p[0]) * ($dy <=> 0);
...
140       $xb = sp($coo[1]);
141       $yb = sp($co2);
142       if (not $thicknessflag) {
143         $xtex .= "\\linethickness{0.8pt}\n";
144         $thicknessflag = 1;
145       }
146       $xtex .= "\\put(".$xb.",".$yb."){\\vector(".$psx.",".$psy.
   "){".$len."}}\n";
```

For the generation of unbroken line segments respectively of complete arrows with the package `pict2e`, the subroutine `best` is provided which yields the best possible numerators and denominators for the rational approximations of the slope with the aid of a continued fraction algorithm. Since, particularly, both components are relatively prime, it is practically no restraint that they must lie in the interval $[-1000, 1000]$.

With the two-line subroutine `sp`, the numbers which Cabri Geometry hands over to the PostScript file with six decimal places, as well as all other decimal numbers appearing in the LaTeX `picture` environment, are rounded off to three places with removal of all concluding zeros.

```
150 foreach (@poly) {
151   @po = split;
152   $p0 = $po[0];
153   $pon = $#po;
154   if (($p0 eq $red) or ($p0 eq $purple) or ($p0 eq $darkbrown)
155     or ($p0 eq $orange) or ($p0 eq $brown)) {
```

The evaluation of the data of the array `@poly` depends on the colour and on the index `$pon=$#po` of the last element.

```
156     if ($pon == 6) {
157       tri($p0,$po[1],$po[2],$po[3],$po[4],$po[5],$po[6]);
```

If `$pon = 6`, then the subroutine `tri` is called which settles the filling, dotting and hatching of triangles .

```
159     elsif ($pon == 8) {
160       ($p0,$u1,$v1,$u2,$v2,$u3,$v3,$u4,$v4) = @po;
161       $s1 =  abs($u1 - $u4) + abs($u2 - $u3) + abs($v1 - $v2) +
   abs($v3 -$v4);
162       $s2 =  abs($u1 - $u2) + abs($u3 - $u4) + abs($v1 - $v4) +
   abs($v2 -$v3);
163       if (($s1 < 4.0E-6) or ($s2 < 4.0E-6)) {
 ...
166         rect($p0,$u1,$v1,$u2,$v2,$u3,$v3,$u4,$v4);
167       }
168       else {
 ...
173         tri($p0,$u1,$v1,$u2,$v2,$u3,$v3);
174         tri($p0,$u1,$v1,$u3,$v3,$u4,$v4);
175       }
176     }
177     elsif ($pon > 8) {
...
179       for (my $j = 3; $j <= $pon - 3; $j += 2) {
...
181         tri($p0,$po[1],$po[2],$po[$j],$po[$j + 1],$po[$j +
   2],$po[$j + 3]);
```

In case of `$pon = 8`, rectangles with sides parallel to the axes are treated with the aid of the subroutine `rect`. This is simpler than the procedure for all other polygons which first have to be cut up in triangles.

```
191   elsif ($p0 eq $blue) {
192     if ($pon == 4) {
193 # Text marker and Bézier curves:
194     $coun++;
...
201     $mtex .= "\\put(".$po1.",".$po2."){".$coun."}\n";
202       }
203     elsif ($pon == 6) {
204       $xtex .= "%Quadratic  Bézier curve\n";
205       qbez($po[1],$po[2],$po[3],$po[4],$po[5],$po[6]);
206     }
207     elsif ($pon == 8) {
208       $xtex .= "%Cubic  Bézier curve\n";
209       cbez($po[1],$po[2],$po[3],$po[4],$po[5],$po[6],$po[7],
   $po[8]);
```

With colour `$blue`, `$pon = 4` yields the text markers, whereas `$pon = 6` and `$pon = 8` lead to the call of the subroutines `qbez` for quadratic Bézier curves and `cbez` for cubic Bézier curves respectively.

```
214 # Arrows
215 foreach (@vector) {
216   @ve = split;
217   if ($ve[0] eq $darkgreen) {
...
235     tri($red,$vu0,$vu1,$vu2,$vu3,$ve[5],$ve[6]);
236     tri($red,$vu0,$vu1,$vu6,$vu7,$ve[5],$ve[6]);
```

Here, as already mentioned above, for arrows with a dotted arrow line (and always for arrows in `CABebez.pl`) the arrowhead is put together joining two filled triangles.

```
258 # Circles, halves and quarters of circles
259 $cflag = 1;
260 $aflag = 1;
261 foreach (@circle) {
262   @po = split;
263   $p0 = $po[0];
264   if ($p0 ne $yellow) {
265     $po[2] = (-1) * $po[2];
266     $di = 2 * $po[3];
```

```
267     if ($po[4] > 1.0E-3 or abs($po[5] - 360) > 1.0E-3) {
268       if ($aflag) {
269         $xtex .= "%Arcs\n";
270         $aflag = 0;
271       }
272       $arce = ($po[4] > 0) ? 360 - $po[4] : 0;
273       $arcb = ($po[5] > 0) ? 360 - $po[5] : 0;
274       $darc = $arce - $arcb;
275       if ($darc < 0) {$darc += 360}
276       $quar = int($darc / 90);
277       if ($quar > 0) {
278         for (my $k = 1; $k <= $quar; $k++) {
279           arc($p0,$po[1],$po[2],$po[3],$arcb,$arcb + 90);
280           $arcb += 90;
281           if ($arcb > 360) {$arcb -= 360}
282         }
283       }
284       if ($darc > $quar * 90) {
285         arc($p0,$po[1],$po[2],$po[3],$arcb,$arce);
286       }
287     }
288     else {
289       if ($cflag) {
290       $xtex .= "%Circles, halves and quarters of circles\n";
291       $cflag = 0;
292       }
293       if ($p0 eq $navy) {
294         $xtex .= "\\put(".$po[1].",".$po[2]."){\\circle{".$di.
   "}}\n";
...
299       }
300       elsif ($p0 eq $purple) {
301         $xtex .= "\\put(".$po[1].",".$po[2]."){\\oval[".$di."]
   (".$di.",".$di.")[l]}\n";
...
340       elsif ($p0 eq $darkgreen) {
341         $r = $po[3];
342         $le = int(2 * $pointf * $ul * $Pi * $r);
343         $xtex .= "\\cCircle[".$le."](".$po[1].",".$po[2].")
   {".$r."}[f]\n";
```

Circular arcs are cut up to quadrants of a circle and to a shorter remaining arc. In `CABebez.pl` the subroutine `quart` approximates all quadrants of a circle (and with that, also circles and semicircles) by a cubic Bézier curve. With `pict2e`, the plot commands `\oval` and `\circle` are available without restraints. For all

remaining circular arcs the subroutine `arc` supplies the approximation by a cubic Bézier curve. The plot command `\cCircle` of the package `ebezier` is used to get dotted circles in `CABpict.pl`.

```
379 # Frame
380 if ($xtex . $mtex  ne "") {
381 $xtex = "\\documentclass{article}\n\\usepackage{ebezier}\n".
382 "\\usepackage[pdftex,pstarrows]{pict2e}\n\n
   \\begin{document}\n\n".
383 "\\setlength{\\unitlength}{".$ul."pt}\n".
384 "\\begin{picture}(".ceil(($xmax - $xmin)).",".
385 ceil(($ymax - $ymin)).")(".floor($xmin).",".floor($ymin).")\n".
386 "\\linethickness{0.8pt}\n"."\\thicklines\n".$xtex;
387 $xtex .= $mtex."\\end{picture}\n\n\\end{document}";
388 }
389 print $xtex."\n";
```

```
342: if ($bmax > 500) {
343:   $bmax = 100 * ceil($bmax / 100);
344:   $btex .= "\\renewcommand{\\qbeziermax}{".$bmax."}\n";
345: }
346: if ($xtex . $mtex  ne "") {
347: $xtex = $btex."\\begin{document}\n\n\\setlength{\\unitlength}{".
348: $ul."pt}\n"."\\begin{picture}(".ceil(($xmax - $xmin)).",".
349: ceil(($ymax - $ymin)).")(".floor($xmin).",".floor($ymin).")\n".
```

Here the different parts of the character string `$xtex` are united and written into the Perl output window by the `print` command.

Since the subsequently assembled subroutines can be used in other versions largely unchanged, now follow only comments which explain the underlying theory. In the main part of `best`, the entered decimal number is transformed into a continued fraction. Then, among all principal and intermediate convergents which are “best approximations”, the one with the greatest possible numerator or denominator is determined.

```
536 # Triangles
537 sub tri {
538 my ($q0,$qx1,$qy1,$qx2,$qy2,$qx3,$qy3) = @_;
...
545   if ($q0 eq $red) {
546 # Filled triangle
547     %ha = ($qx1,$qy1,$qx2+1e-07,$qy2+1e-07,$qx3+2e-07,
  $qy3+2e-07);
```

```
548     @hb = ();
549     @hc = ();
550     foreach (sort { $ha{$a} <=> $ha{$b} } keys %ha) {
551       $hb[++$#hb] = $_;
552       $hc[++$#hc] = $ha{$_};
553     }
554     ($qx1,$qx2,$qx3) = @hb;
555     ($qy1,$qy2,$qy3) = @hc;
```

Each of the three parts of `tri` begins with sorting out corners: For filling and dotting, they are ordered according to the size of the y-coordinates and for hatching, the order depends on the difference of both coordinates. For the sorting method used by Perl for the data type "hash", it should be taken into account that in our case, each of the two associated arrays must consist of different elements. Here, this is achieved by changing the seventh place behind the decimal point.

If, after sorting, one imagines a straight line passing through the first and third corner, linear algebra yields the half-plane bordered by the straight line and containing the second corner. Then, starting with the connecting line segment of the first and third corner and with distinction of cases concerning the second point, the objects which have to be inserted are constructed.

The intermediate points of quadratic and cubic Bézier curves needed for their bounding box are calculated with the formulas of Section 4. The coefficients of cubic Bézier curves which approximate parts of quadrants of circles in the subroutine `arc` are derived in [2] (Pages 11 f).

8. The `picture` environment of LaTeX

Using the capacity of geometry software or of drawing programs, also further advantages of the LaTeX `picture` environment compared with the `\special` command come into effect:

- All graphic data are integrated in the TeX file. Particularly, no graphic files can get lost during dissemination.
- LaTeX proves its flexibility mainly with the positioning using `minipages` or `floats` (sliding objects). Since the filter programs yield the size of the bounding box as well as the offset value (for the upper left corner), each figure can directly be placed in a `minipage`. Using that it can be moved horizontally

and vertically, for example, applying the commands `\hspace` and `\raisebox` (see Figures 1 to 5).

- The final processing of figures can be done in the usual cycle edit-typeset-preview.
- Even in documents with numerous complicated figures, memory requirements can well be estimated and, particularly, applying `pict2e`, they are surprisingly low. But also with the `ebezier` package, in BigTeX versions no lack of memory has to be expected.
- If PDF files are generated from LaTeX productions, the figures are taken over automatically.

The completion of the LaTeX `picture` environment with commands from the `color` package mentioned in Section 3 is possible only with restraints. Due to a technical difficulty, TeX possibly inserts vertical space at each change of colour (see [3], Page 6). Therefore, without complicated corrections, only figures can be coloured for which all connected parts have the same colour.

The development of filter programs for LaTeX was initiated by the author's wish to take over about 80 figures created with the drawing program STAD on Atari computers into a book designed with LaTeX on Macintosh computers. The figures were traced with the drawing program ClarisDraw, and the PostScript versions were transformed to LaTeX `picture` environments with an AppleScript program. This first functioning PostScript filter program can be found in [6] in the section 'Programme'.

References

[1] Adobe, Systems Incorporated: PostScript Language Reference Manual. Addison-Wesley, 2nd edition, 1995.

[2] Bachmaier, Gerhard A.: Using `ebezier`. Package in CTAN:`/macros/latex/contrib/ebezier`, 2002.

[3] Carlisle, David P.: Packages in the 'graphics' bundle. CTAN:`/macros/latex/required/graphics/grfguide.pdf`, 1999/2004.

[4] Gäßlein, Hubert and Niepraschk, Rolf: The `pict2e`-package. Package in CTAN:`/macros/latex/contrib/pict2e/pict2e.dtx`, 2003.

[5] Goossens, Michel, Rahtz, Sebastian and Mittelbach, Frank: The LaTeX Graphics Companion. Addison-Wesley, Reading MA, 1997.

[6] Möller, Herbert: *Mathcompass* (`http://wwwmath1.uni-muenster.de/u/mollerh`).

[7] Nickalls, Richard W.D.: MathsPIC. CTAN:`/graphics/pictex/mathspic`, 1999.

[8] Nickalls, Richard W.D. and Apostolos Syropoulos: $mathsPIC_{Perl}$. CTAN:`/graphics/pictex/mathspic/Perl`, 2005.

[9] Wichura, Michael J.: The PICTeX Manual, 1992. Package in CTAN:`/graphics/pictex`.