Macros

DDA Methods in TeX

David Salomon

Several macros are presented here that use DDA methods to generate lines, circles, and ellipses. They are all based on the idea that a curve can be drawn in TeX by moving a dot in small steps and repeatedly typesetting it. This idea was originally suggested by Hendrickson [1] for straight lines, and extended by Cameron [2] for wiggly lines. LaTeX users also have line and circle macros available, but those are limited to certain slopes and diameters.

The macros presented here generate lines, circles, and ellipses, using DDA (Digital Differential Analyzer) methods. DDA is a general name for methods that generate geometric shapes using simple arithmetic operations, and integers. No multiplication, division, square root, or floating-point numbers are used. Typically, a DDA method works by moving along the curve in small steps, calculating the coordinates of the next point (x_{i+1}, y_{i+1}) either as simple functions of the current point (x_i, y_i), or using a parametric representation of the curve. Thus either $x_{i+1} = f(x_i, y_i)$ and $y_{i+1} = g(x_i, y_i)$; or $(x, y) = f(\phi)$.

The first macro uses the *Quadrantal DDA* method [3] to produce straight lines of any slope. The second macro is a TeX implementation of the *Octantal DDA* method [3], which is somewhat more involved but produces finer lines. The third macro implements *Bresenham's algorithm* [4] for circles; and the last two macros, for ellipses, are based on the parametric equation of these curves. In most of the cases above, the precise shape of the curve depends on the size of the basic step, which is the value of the `\dimen` variable `\step`. It is recommended to first experiment with the macros using `\step=1pt`, just to see how the dot is moved for any given curve. For production purposes, however, it is better to set `\step=.25pt`, which produces a small enough step size such that, in a 300 dpi output, curves look pretty smooth. For higher resolution outputs, the step size should be made even smaller. Unfortunately, making the step size too small, or generating long curves, may result in the dreaded (and, alas, familiar) message:

```
! TeX capacity exceeded,
    sorry [main memory size=65536].
```

The ellipse macros have another potential problem. Large ellipses may cause an `arithmetic overflow` message, due to TeX's limited capacity.

The Quadrantal DDA Method

Macro `\quadr` typesets a slanted line by using the quadrantal DDA method. It works in any mode and does not move the reference point.

The macro has 2 parameters, Δx and Δy, which are the horizontal and vertical projections of the line, respectively. Since the line starts at the current reference point, the two parameters can also be viewed as the coordinates of the endpoint of the line (relative to the reference point). The parameters can be specified in any valid TeX dimension, so expansions such as

```
\quadr -13pt 5in
\quadr 25pc -5mm
\quadr 3cc 3dd
```

are all valid. Note the percent signs '`%`' at the end of certain macro lines. They are important because TeX converts an end of line to a space, but we don't want such spaces to get typeset (try eliminating some of the '`%`' to see what happens).

To understand the principle of the quadrantal DDA method, consider the case where both Δx and Δy are positive. The line should go up and to the right from the current reference point. The method works by typesetting a dot at the reference point, then moving it, by the basic step, either up or to the right (but not in both directions), typesetting it again, and looping, until the dot has been moved a distance of Δx in the x direction, and a distance of Δy in the y direction.

Figure 1 shows two such lines, one almost horizontal and the other, at 45°. Each dot has been magnified to a small box. Note how the principle, of moving either to the right or up, creates the line as a number of *overlapping* segments. This causes the line to appear thicker than it should be. If either Δx or Δy is negative, the dot has to be moved to the left or down. Our algorithm thus has four parts — macro `\doloopA` is used if $\Delta x \geq 0$ and $\Delta y \geq 0$ ($0° \leq$ slope $\leq 90°$, the first quadrant); `\doloopB` is used if $\Delta x < 0$ and $\Delta y \geq 0$ ($90° <$ slope $\leq 180°$, the second quadrant); and so on.

The decision in what direction to move is based on the value of the `\count` variable `\diff`. `\diff` is initially set to $-0.5\Delta x$ and is either decremented by Δx (if a decision is made to move the dot up), or incremented by Δy (if the dot is to be moved to the right). By the time the dot gets all the way to the end point of the line, the ratio between the number of times it has been moved up and the number of times it has been moved to the right is $\Delta y/\Delta x$, which is the slope of the line. The algorithm for the first quadrant is therefore:

```
$x := 0$; $y := 0$; $diff := -\Delta x/2$;
repeat
  plot$(x, y)$;
  if $diff > 0$
    $y := y + 1$; $diff := diff - \Delta x$;
  else
    $x := x + 1$; $diff := diff + \Delta y$;
until $x = \Delta x$ & $y = \Delta y$;
```

A simple example is a line with $\Delta x = 6$ and $\Delta y = 2$. The algorithm above iterates 9 times as summarized in the table.

			diff	
step	x	y	before	after
1	0	0	-3	$-3+2$
2	1	0	-1	$-1+2$
3	1	1	1	$1-6$
4	2	1	-5	$-5+2$
5	3	1	-3	$-3+2$
6	4	1	-1	$-1+2$
7	4	2	1	$1-6$
8	5	2	-5	$-5+2$
9	6	2	-3	

For more information on this method, see reference [3].

Macro \quadr uses the 4 macros \doloopA, ..., \doloopD. However, only one of them is expanded, depending on the slope of the desired line. Macro \point typesets a dot by placing it in a box of zero dimensions (see page 389 of [5]). Macro \point and a number of registers are used by both the quadrantal and octantal methods; a user creating a file of macros for either method must include the following code.

```
\newdimen\deltax \newdimen\deltay
\newcount\diff
\newdimen\xstep \newdimen\ystep
\newdimen\step
\newif\ifmore
%
\def\point#1#2{% keep this percent sign!
  \vbox to0pt{\kern-#2
    \hbox to0pt{\kern#1.\hss}\vss}%
  \ifvmode\nointerlineskip\fi}
```

Exercise 1: Look carefully at the way macro \point generates boxes. What is the depth of those boxes?

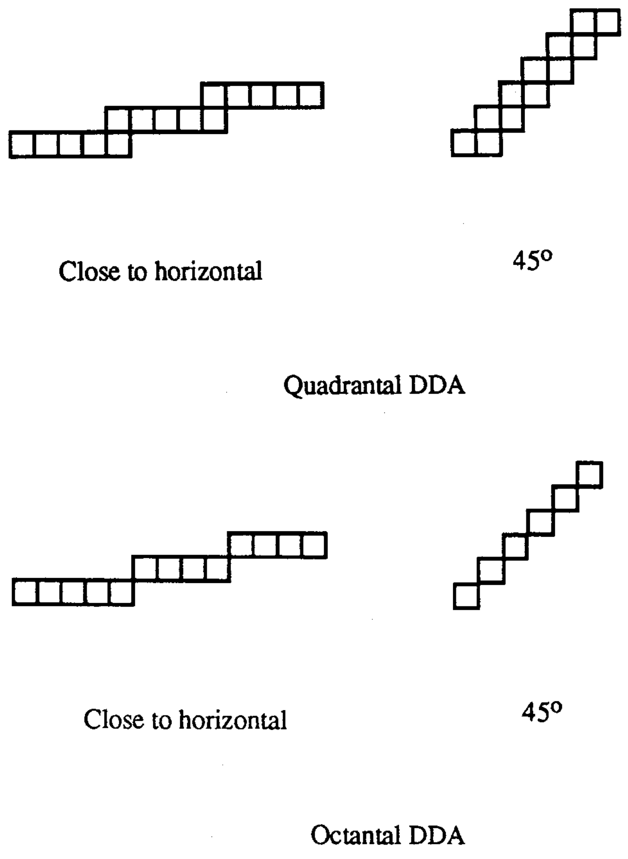


Figure 1. Details of Quadrantal and Octantal Lines

Note that \nointerlineskips are inserted between the dots when TEX is in vertical mode. This avoids the interline glue which otherwise is automatically generated. As a result, macro \quadr does not move the reference point and, after each expansion, the user should decide whether to move it, and by how much.

```
\def\quadr#1 #2 {% keep this % sign!
\deltax=#1 \deltay=#2
\xstep=0pt \ystep=0pt
\ifdim\deltax<0pt
  \ifdim\deltay<0pt \doloopC
  \else \doloopB    \fi
\else
  \ifdim\deltay<0pt \doloopD
  \else  \doloopA  \fi
\fi}                    % end of macro quadr
\def\doloopA{%
  \ifdim\deltax>\deltay \diff=-\deltax
  \else \diff=\deltay \fi
  \divide\diff by 2
\loop
\ifnum\diff>0
  \advance\ystep by \step
  \advance\diff by-\deltax
\else
  \advance\xstep by \step
  \advance\diff by \deltay
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep<\deltax \moretrue\fi
\ifdim\ystep<\deltay \moretrue\fi
\ifmore\repeat}
      % end of loop for 1st quadrant
%
\def\doloopB{%
  \ifdim-\deltax>\deltay \diff=\deltax
  \else \diff=\deltay \fi
  \divide\diff by 2
\loop
\ifnum\diff>0
  \advance\ystep by \step
  \advance\diff by \deltax
\else
  \advance\xstep by-\step
  \advance\diff by \deltay
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep>\deltax \moretrue\fi
\ifdim\ystep<\deltay \moretrue\fi
\ifmore\repeat}
             % end of loop for 2nd quadrant
%
\def\doloopC{%
  \ifdim-\deltax>-\deltay \diff=\deltax
  \else \diff=-\deltay \fi
  \divide\diff by 2
\loop
\ifnum\diff>0
  \advance\ystep by-\step
  \advance\diff by \deltax
\else
  \advance\xstep by-\step
  \advance\diff by-\deltay
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep>\deltax \moretrue\fi
\ifdim\ystep>\deltay \moretrue\fi
\ifmore\repeat}
             % end of loop for 3rd quadrant
%
\def\doloopD{%
  \ifdim\deltax>-\deltay \diff=-\deltax
  \else \diff=-\deltay \fi
  \divide\diff by 2
\loop
\ifnum\diff>0
  \advance\ystep by-\step
  \advance\diff by-\deltax
\else
  \advance\xstep by \step
  \advance\diff by-\deltay
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep<\deltax \moretrue\fi
\ifdim\ystep>\deltay \moretrue\fi
\ifmore\repeat}
             % end of loop for 4th quadrant
```

The Octantal DDA Method

Macro \octnt typesets a slanted line using *octantal* DDA, a method very similar to quadrantal DDA. The main difference is the way the dot is moved between repeated typesettings. If the line is close to horizontal (its slope is between 0° and 45°) the dot is moved either to the right, or diagonally (up and to the right). If the line is close to vertical ($\Delta y > \Delta x$ or the slope is between 45° and 90°), the dot is moved either up or diagonally. If either Δx or Δy is negative, the directions are changed accordingly.

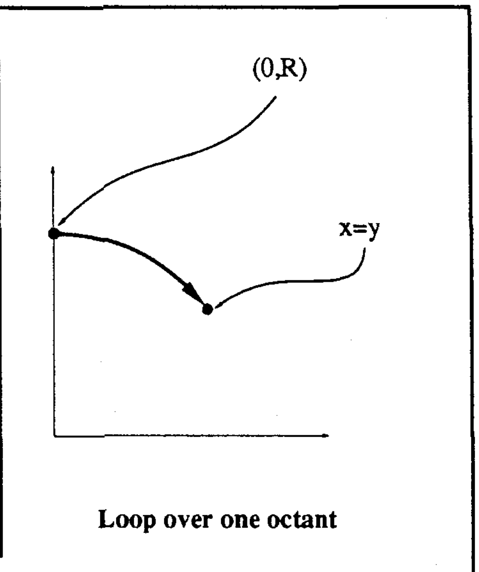
Fig. 1 shows the way lines appear in this method. The line that is close to horizontal is made of several non-overlapping segments; the 45° line consists of dots laid diagonally. These lines are finer than the quadrantal lines since they consist of fewer dots.

Because of the rules above, the algorithm should distinguish eight orientations of the lines, or eight ranges of the slope (hence the name *octantal*). The main macro, \octnt, does exactly that. However, the range 0°–45° (octant 1) is similar to the range 315°–360° (octant 8), so they are both handled by macro \loopA. Octants 2, 3 (45°–90°, 90°–135°) are handled by macro \loopB, and so on. We thus end up with just four loop macros, instead of eight. Dots are typeset by the same macro, \point, used for lines drawn by the quadrantal method.

```
$x := 0$; $y := 0$; $diff := -\Delta x/2$;
**repeat**
  plot$(x, y)$;
  **if** $diff > 0$
    $y := y + 1$; $x := x + 1$;
```

```
x:=0; y:=R; d:=3-2R;
while x<y do
plot8(x,y);
if d>0 then
          d:=d+4(x-y)+10;
          y:=y-1;
        else
          d:=d+4x+6;
x:=x+1;
end while
if x=y then plot8(x,y);
end;
```

Final Program



Loop over one octant

Figure 2. Bresenham's Algorithm for a Circle

```
    $diff := diff - \Delta x + \Delta y$;
  else
    $x := x + 1$; $diff := diff + \Delta y$;
until $x := \Delta x$;
```

And we illustrate the method with the previous example; a line with $\Delta x = 6$ and $\Delta y = 2$. This time the algorithm iterates only 7 times, producing a finer line.

			diff	
step	x	y	before	after
1	0	0	-3	$-3+2$
2	1	0	-1	$-1+2$
3	2	0	1	$1-6+2$
4	3	1	-3	$-3+2$
5	4	1	-1	$-1+2$
6	5	1	1	$-1-6+2$
7	6	2	-3	

```
\newdimen\Absx \newdimen\Absy
\newdimen\Xstep \newdimen\Ystep
%
\def\octnt#1 #2 {% keep this percent sign
\deltax=#1 \deltay=#2
\xstep=0pt \ystep=0pt
\ifdim\deltax<0pt \Absx=-\deltax
  \else \Absx=\deltax \fi
\ifdim\deltay<0pt \Absy=-\deltay
  \else \Absy=\deltay \fi
\ifdim\deltax<0pt
  \ifdim\deltay<0pt
    \Xstep=-\step \Ystep=-\step
    \ifdim\Absx>\Absy \loopD
    \else \loopC \fi
%           octants 5 (loopD) & 6 (loopC)
  \else
    \Xstep=-\step \Ystep=\step
    \ifdim\Absx>\Absy \loopD
    \else \loopB \fi
%           octants 4 (loopD) & 3 (loopB)
  \fi
\else
  \ifdim\deltay<0pt
    \Xstep=\step \Ystep=-\step
    \ifdim\Absx>\Absy \loopA
    \else \loopC \fi
%           octants 8 (loopA) & 7 (loopC)
  \else
    \Xstep=\step \Ystep=\step
    \ifdim\Absx>\Absy \loopA
    \else \loopB \fi
%           octants 1 (loopA) & 2 (loopB)
  \fi
\fi}              % end of macro \octnt
%
\def\stepx{\advance\xstep by \Xstep
  \advance\diff by \Absy}
\def\stepy{\advance\ystep by \Ystep
  \advance\diff by-\Absx}
%
\def\loopA{%    loop for octants 1 & 8
\diff=-\Absx \divide\diff by 2
\loop
\ifnum\diff>0
     \stepx \stepy
\else \stepx
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep<\deltax \moretrue\fi
\ifmore\repeat}
        % end of loop for octants 1 & 8
%
\def\loopB{%   loop for octants 2 & 3
\diff=\Absy \divide\diff by 2
\ifdim\Absx=\Absy \diff=0 \fi
\loop
\ifnum\diff>0
      \stepy
\else \stepx \stepy
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\ystep<\deltay \moretrue\fi
\ifmore\repeat}
          % end of loop for octants 2 & 3
%
\def\loopC{%     loop for octants 6 & 7
\diff=\Absy \divide\diff by 2
\ifdim\Absx=\Absy \diff=0 \fi
\loop
\ifnum\diff>0
      \stepy
\else \stepx \stepy
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\ystep>\deltay \moretrue\fi
\ifmore\repeat}
          % end of loop for octants 6 & 7
%
\def\loopD{%     loop for octants 4 & 5
\diff=-\Absx \divide\diff by 2
\loop
\ifnum\diff>0
      \stepx \stepy
\else \stepx
\fi
\point{\xstep}{\ystep}%
\morefalse
\ifdim\xstep>\deltax \moretrue\fi
\ifmore\repeat}
          % end of loop for octants 4 & 5
```

Note that \loopA (octants 1 and 8) repeats while the x-coordinate of the dot is $< \Delta x$. \loopD (octants 4 and 5, where x and Δx are negative), however, repeats while $x > \Delta x$. This works since in those octants the line is closer to horizontal. \loopB and \loopC, where lines are close to vertical, are similar but compare y and Δy.

The Bresenham-Michener DDA Method for Circles

Because of the high symmetry of a circle, it is a particularly easy figure to draw (See [6] for a number of circle drawing methods). The method used here is efficient since it uses only integers and requires only addition, subtraction, and a multiplication by 4. When this method is implemented as a computer program, the multiplication by 4 is usually replaced by a shift. TeX, however, cannot shift numbers. The method is described here in two stages. First the basic idea (see algorithm in Fig. 2) is outlined; next, the TeX implementation is explained.

The basic idea is to draw a circle of radius R, centered around the origin, by starting at the top of the circle (point $(0, R)$) and moving, in small steps, along one octant of the circle. Because of the symmetry of a circle, each time a point (x, y) is calculated on one octant, seven more points — on the seven other octants — can be calculated, which correspond to the original point. They are: $(-x, y)$, $(x, -y)$, $(-x, -y)$, (y, x), $(-y, x)$, $(y, -x)$, and $(-y, -x)$. The algorithm is a simple loop that starts at point $(x, y) = (0, R)$ and continues while $x < y$ (i.e., over one octant). Each time through the loop, the current point (plus the seven corresponding points) is typeset, and the algorithm moves to the next point by incrementing the x coordinate by \step and, from time to time, also decrementing the y coordinate (by the same \step). Variable \step has to be assigned a value before \circle is expanded.

The only decision that has to be made in each iteration is whether or not to decrement the y coordinate. This decision involves the auxiliary \dimen variable \d whose sign determines the action taken. If \d is non-negative then y is decremented. Each time through the loop \d is updated. The details of updating \d can be found in references [4, 6] or can be obtained by writing to this author.

The TeX implementation presented here builds the circle centered on the reference point. The reference point itself is not moved and, after each expansion of the macro, the user may want to move it explicitly, using appropriate skip commands. Macro \circle is a simple \loop construct that expands a plot macro to plot the current point (actually, eight points), and then calculates the coordinates of the next point. Like the macro \point used to plot slanted lines, macro \plot typesets a dot enclosed in boxes of zero dimensions (this is why the reference point is not affected), and in vertical mode, inserts `\nointerlineskip` between boxes to eliminate unwanted interline glue. Again note the percent signs '%' at the end of certain source lines. They are important and have been mentioned earlier.

```
\newdimen\xc \newdimen\yc
\newdimen\d  \newdimen\tc
\newdimen\unitc
%
\def\circle#1{\unitc=1pt
  \xc=0pt \yc=#1 \d=3\unitc
  \advance\d by-2\yc
  \loop
    \ploteight
    \ifdim\d>0pt
      \tc=\xc \advance\tc by-\yc
      \multiply\tc by 4
      \advance\tc by 10\unitc
      \advance\d by \tc
      \advance\yc by-\step
    \else
      \tc=4\xc
      \advance\tc by 6\unitc
      \advance\d by \tc
    \fi
    \advance\xc by \step
  \ifdim\xc<\yc \repeat}
%
\def\ploteight{%
  \plot{\xc}{\yc}\plot{-\xc}{\yc}%
  \plot{\xc}{-\yc}\plot{-\xc}{-\yc}%
  \plot{\yc}{\xc}\plot{-\yc}{\xc}%
  \plot{\yc}{-\xc}\plot{-\yc}{-\xc}}
%
\def\plot#1#2{%
  \vbox to0pt{\kern#2
    \hbox to0pt{\kern#1.\hss}\vss}%
  \ifvmode\nointerlineskip\fi}
```

Drawing Ellipses

The ellipse macros below accept, as parameters, the semimajor and the semiminor ellipse axes, measured in pt. The first macro generates a canonical ellipse (centered on the reference point with a horizontal major axis); the second one generates an ellipse tilted clockwise θ degrees.

Bresenham's method can be generalized to an ellipse (Try to do it! This is exercise 2.), but this does not give good results because of the symmetry of the ellipse, which is not as high as that of a circle. In the case of a circle, it is enough to calculate one octant and copy it over to the other seven. In the case of an ellipse, one quadrant, at least, has to be calculated. Bresenham's method is based on looping in equal steps of x, and this produces good results in the first octant, since that octant has a small slope and does not deviate much from a horizontal line. Looping in equal steps of x over a quadrant, however, produces dots that are too widely spaced at the end of the quadrant (Fig. 3a), where the ellipse has a large slope.

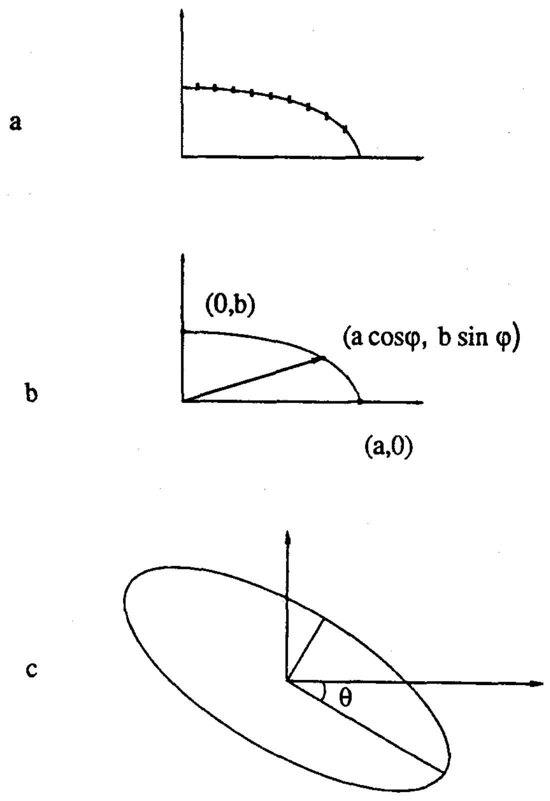


Figure 3. Ellipses

The method used here to draw an ellipse is well known [7, 8] and is based on the parametric representation of the ellipse:

$$x = a\cos\phi; \quad y = b\sin\phi. \qquad \phi = 0\ldots360^\circ \quad (1)$$

where a is the semimajor axis and b, the semiminor one (Fig. 3b). The parameter ϕ is varied (in small steps of $d\phi$), over a quarter of the ellipse, from point (a,0), ($\phi = 0$), to point (0,b), ($\phi = 90^\circ$).

An important property of the algorithm is that varying ϕ in fixed steps of $d\phi$ moves the dot along the ellipse in steps that cover variable perimeter sizes. Initially, around point (a,0), the step size is small, which is appropriate for that region, where the ellipse has a large slope. As we move along the quadrant toward the final point (0,b), the step size covers larger perimeter increments, again appropriate for this region, where the slope gets smaller.

To demonstrate this property, we derive the differential of Eq. 1.

$$dx = -a\sin\phi\,d\phi; \quad dy = b\cos\phi\,d\phi.$$

For the initial steps, where ϕ is close to zero, $dy \approx b\,d\phi$ and dx is close to zero. Toward the end, where ϕ is close to 90°, $|dx| \approx a\,d\phi$ and $dy \approx 0$. The perimeter increment is thus initially close to $b\,d\phi$ and gets larger as it approaches $a\,d\phi$. Also, the ratio between the initial and final perimeter increments is approximately b/a, which is the ratio of the two axes of the ellipse. If $a = b$, the perimeter increment is fixed, which is appropriate for a circle.

Our method uses the elementary trigonometric identities:

$$\begin{aligned}\sin(x+y) &= \sin x\cos y + \cos x\sin y;\\ \cos(x+y) &= \cos x\cos y - \sin x\sin y.\end{aligned} \tag{2}$$

Using Eq. 1, we start with $\phi = 0$ and get:

$$\begin{aligned}x_0 &= a\cos 0 = a; & y_0 &= b\sin 0 = 0.\\ x_1 &= a\cos(d\phi); & y_1 &= b\sin(d\phi).\\ x_2 &= a\cos(2\,d\phi); & y_2 &= b\sin(2\,d\phi).\end{aligned}$$

And, in general

$$x_i = a\cos(i\,d\phi) = a\times A_i; \quad y_i = b\sin(i\,d\phi) = b\times B_i.$$

The DDA nature of the algorithm stems from the fact that we can eliminate the need for calculating $\sin(i\,d\phi)$, $\cos(i\,d\phi)$ for every value of i. Using Eq. 2, it is possible to express both A_i, B_i as functions of A_{i-1}, B_{i-1} with the result that only $\sin(d\phi)$, $\cos(d\phi)$ need be known.

$$A_i = \cos(i\,d\phi) = \cos((i-2)d\phi + 2d\phi)$$

using Eq. 2 yields

$$\begin{aligned}A_i = {}&\cos((i-2)d\phi)\cos(2d\phi)\\ &- \sin((i-2)d\phi)\sin(2d\phi);\end{aligned}$$

using Eq. 2 again

$$\begin{aligned}A_i = {}&\cos((i-2)d\phi)[\cos^2(d\phi) - \sin^2(d\phi)]\\ &- 2\sin((i-2)d\phi)\sin(d\phi)\cos(d\phi);\end{aligned}$$

adding and subtracting the same term

$$\begin{aligned}A_i = {}&\cos((i-2)d\phi)\cos^2(d\phi)\\ &- \sin((i-2)d\phi)\sin(d\phi)\cos(d\phi)\\ &- \sin((i-2)d\phi)\cos(d\phi)\sin(d\phi)\\ &- \cos((i-2)d\phi)\sin^2(d\phi)\\ = {}&A_{i-1}\cos(d\phi) - B_{i-1}\sin(d\phi).\end{aligned} \tag{3}$$

And, similarly,

$$B_i = B_{i-1}\cos(d\phi) + A_{i-1}\sin(d\phi). \tag{4}$$

The initial values are $A_0 = \cos 0 = 1$, $B_0 = \sin 0 = 0$. Our algorithm can now be expressed as:

```
$A := 1$; $B := 0$; $C := \cos(d\phi)$; $S := \sin(d\phi)$;
$x := a$; $y := 0$;
loop
  plot $(x, y)$ plus three symmetric points
  $T := A\times C - B\times S$;
  $B := B\times C + A\times S$;
  $A := T$;
  $x := a\times A$; $y := b\times B$;
while $x > 0$;
```

This algorithm involves multiplications, and is therefore considerably slower than Bresenham's, but then a circle is just a special case of an ellipse. Needless to say, the ellipse macro below can be used to generate circles. The macro is a TeX implementation of the rules above, with two exceptions:

1. In principle, the user should supply a value for $d\phi$ and the macro should calculate $\sin(d\phi)$ and $\cos(d\phi)$. However, since those calculations involve fractions, they have to be done, in TeX, with scaled numbers, which is time consuming. As a result, three pairs of $\sin(d\phi)$ and $\cos(d\phi)$ are built into the macro, corresponding to $d\phi$ values of $2\pi/120$, $2\pi/240$ and $2\pi/480$. Those values were selected experimentally, to produce smooth ellipses on a 300 dpi output. On higher resolution output devices, smaller values should be tried, which may result in finer curves. The first pair generates an ellipse by typesetting 120 dots (actually, generating 30 dots and duplicating each 4 times), the second typesets 240 dots and the third, 480. The macro selects one of those pairs, depending on the size of the ellipse.
2. TeX can easily operate on integers but our problem involves real numbers. Such problems are handled in TeX in one of two ways. The first is to use `dimen` variables, which can have non-integer values; the second makes use of scaled integers. Our macro uses the second choice and scales all numbers by a `\scalefactor` of 10000.

The following registers and macro are common to both ordinary and tilted ellipses. Once again, note the similarity of `\plotu` to the earlier `\point` and `\plot` macros.

```
1. \newcount\a \newcount\A
2. \newcount\b \newcount\B \newcount\T
3. \newcount\c \newcount\C
```

```
\newcount\s \newcount\S \newcount\t
\newcount\x \newcount\y
\newcount\scalefactor \scalefactor=10000
\newdimen\unit
\unit=1pt \divide\unit by \scalefactor
%
\def\plotu#1#2{%
  \vbox to0pt{\kern#2\unit
    \hbox to0pt{\kern#1\unit.\hss}\vss}%
  \ifvmode\nointerlineskip\fi}
```

The macro for the ellipse.

```
\def\ellipse#1 #2 {%
  \A=10000 \B=0
  \ifnum#1>#2 \a=#1 \b=#2
  \else \a=#2 \b=#1 \fi
% d\phi is determined according to the
% value of the semimajor axis 'a'.
\ifnum\a<15
  \S=523 \C=9986 % sin & cos of 360/120,
  % correspond to 30 increments of d\phi
\else
  \ifnum\a<40 %over one quarter
    \S=262 \C=9997 %For large ellipses,
                 % here are 60 increments
  \else
    \S=131 \C=9999 % and, for the largest
                     % ones, 120 increments
\fi \fi
%
\x=\a \multiply\x by \scalefactor \y=0
\loop
\plotfour
\T=\B \multiply\T by\S
\t=\A \multiply\t by\C \advance\t by-\T
\T=\A \multiply\T by\S
\multiply\B by\C \advance\B by\T
\divide\B by \scalefactor
\A=\t \divide\A by \scalefactor
\x=\a \multiply\x by\A
\y=\b \multiply\y by\B
\ifnum\x>0 \repeat}
%
\def\plotfour{%
  \plotu{\x}{\y}\plotu{-\x}{\y}%
  \plotu{\x}{-\y}\plotu{-\x}{-\y}}
```

Next, we turn to a tilted ellipse, obtained by rotating the canonical ellipse θ degrees clockwise. Mathematically, rotating a 2D point (x, y) is achieved by multiplying it by the rotation matrix

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

Thus the general ellipse point $(a\cos\phi, b\sin\phi)$ is transformed into

$(a\cos\phi\cos\theta - b\sin\phi\sin\theta, a\cos\phi\sin\theta + b\sin\phi\cos\theta)$.

and the expressions $x := a \times A$, $y := b \times B$, used earlier for the coordinates of the next point now become

$x := a\,A\cos\theta - b\,B\sin\theta \quad y := a\,A\sin\theta + b\,B\cos\theta.$

The algorithm for the tilted ellipse differs from the one for the canonical one in three more points:

1. Since a tilted ellipse is not symmetric with respect to the coordinate axes, macro `\Plotfour` cannot duplicate a point as easily as before. A look at Fig. 3c shows that, for each point (x, y) on one quadrant of the ellipse, point $(-x, -y)$ is on the diagonally opposite quadrant but points $(-x, y)$, $(x, -y)$ are not on the ellipse. The macro should therefore calculate one of these points explicitly, using the expressions
$$u := a\,A\cos\theta + b\,B\sin\theta;$$
$$w := a\,A\sin\theta - b\,B\cos\theta.$$
and plot points (u, w), $(-u, -w)$.
2. The loop for the canonical ellipse is terminated when x reaches zero. This again won't work for the tilted ellipse, so the new macro `\tellipse` uses a new count variable to loop 30, 60, or 120 times, depending on the size of the ellipse. In each iteration, the count variable is decremented and compared to zero.
3. Because of the additional multiplications necessary, numbers cannot be scaled as high as in the previous macro. Trying to scale all numbers with a factor of 10000 causes arithmetic overflow, so reduced scaling is used, resulting in a less precise shape of the ellipse.

The algorithm thus is:

```
$A := 1;\ B := 0;\ C := \cos(d\phi);\ S := \sin(d\phi);$
$x := a\ \cos\theta;\ u := x;$
$y := a\ \sin\theta;\ w := y;$
$count := 30,\ 60,\ \text{or}\ 120$
**loop**
  plot $(x, y), (-x, -y), (u, w), (-u, -w)$
  $T := A \times C - B \times S;$
  $B := B \times C + A \times S;$
  $A := T;$
  $LA := a \cdot A;\ LB := b \cdot B;$
     (with a reduced scale factor)
  $x := LA\cos\theta - LB\sin\theta;$
  $y := LA\sin\theta + LB\cos\theta;$
  $u := LA\cos\theta + LB\sin\theta;$
  $w := LA\sin\theta - LB\cos\theta;$
  $count := count - 1$
  while $count > 0$;
```

and the macro is:

```
\newcount\dphi
\newcount\u \newcount\w
\newcount\ST \newcount\CT
\newcount\LA \newcount\LB
%
\def\tellipse#1 #2 #3 #4 {%
   \A=10000 \B=0
   \d=#3pc \ST=\d
   \divide\ST by 786 %since 1pc=786432sp
   \d=#4pc \CT=\d \divide\CT by 786
   \ifnum#1>#2 \a=#1 \b=#2
   \else \a=#2 \b=#1 \fi
% d\phi is determined according to the
% value of the semimajor axis a.
   \ifnum\a<15
      \S=523 \C=9986 \dphi=31
      % sin, cos of 360/120, for 30
      % increments of d\phi
   \else \ifnum\a<40 % over one quarter.
      \S=262 \C=9997 \dphi=61
      % For large ellipses, here are 60
      % increments
   \else
      \S=131 \C=9999 \dphi=121
      % For the largest ones, 120
      % increments
   \fi \fi
%
\x=\a \multiply\x by\CT
\multiply\x by 10 \u=\x
\y=\a \multiply\y by\ST
\multiply\y by 10 \w=\y
\loop
   \Plotfour
   \T=\B \multiply\T by\S
   \t=\A \multiply\t by\C
   \advance\t by-\T
   \T=\A \multiply\T by\S
   \multiply\B by\C
   \advance\B by\T
   \divide\B by \scalefactor
   \A=\t \divide\A by \scalefactor
%
   \LA=\A \multiply\LA by\a
   \divide\LA by 100
   \LB=\B \multiply\LB by\b
   \divide\LB by 100
%
   \x=\LA \multiply\x by\CT \u=\x
   \T=\LB \multiply\T by\ST
   \advance\x by-\T \divide\x by 10
   \advance\u by \T \divide\u by 10
   \y=\LA \multiply\y by\ST \w=\y
   \T=\LB \multiply\T by\CT
   \advance\y by \T \divide\y by 10
   \advance\w by-\T \divide\w by 10
   \advance\dphi by-1
\ifnum\dphi>0 \repeat}
%
\def\Plotfour{%
   \plotu{\x}{\y}\plotu{\u}{\w}%
   \plotu{-\x}{-\y}\plotu{-\u}{-\w}}
```

This method is considerably slower than the ones for lines and circles because of the multiplications involved. It turns out that, even though not all the multiplications can be eliminated, the method can be made a little more efficient. Reference [9] shows how to modify it to include only four muliplications (and four additions) per iteration. The algorithm is:

```
$CT := \cos\theta$; $ST := \sin\theta$;
$CDP := \cos\, d\phi$; $SDP := \sin\, d\phi$;
$A := CDP + SDP \times ST \times CT \times (a/b - b/a)$;
$B := -SDP\left((b\times ST)^2 + (a\times CT)^2\right)/(a\times b)$;
$C := SDP\left((b\times CT)^2 + (a\times ST)^2\right)/(a\times b)$;
$D := CDP + SDP \times ST \times CT \times (b/a - a/b)$;
$D := D - (C\times B)/A$;
$C := C/A$;
$x := a\times CT$; $y := a\times ST$;
$count := 30,\ 60,\ \text{or}\ 120$;
loop
  plot $(x,y), (-x,-y), (u,w), (-u,-w)$
  $x := x\times A + y\times B$;
  $y := x\times C + y\times D$;
  $count := count - 1$
while $count > 0$;
```

The reader is encouraged to implement this in TEX.

Appendix

The methods described here give reasonably good output on a typical 300dpi printer. Sometimes, however, high quality output is a must. Here is an idea which produces better looking results. It is, unfortunately, slow and is more liable to exceed TEX's capacity.

All the examples above generate the lines and curves by typesetting a period. The period has a small size but is not small enough for high quality results. Using a period from a smaller size font does not help much. It turns out that the width of a period in font cmr10 is `2.77779pt` whereas in font cmr5 it is `2.01392pt`, almost the same size.

To get dots of smaller sizes, we therefore suggest typesetting a rule (specifically, a `\vrule`) instead of a dot. The height and width of a rule can easily be controlled and our experiments show that a rule of dimensions `0.1pt`, combined with a step size of the same dimension, produces fine, smooth lines on a 300dpi laser printer. The only change necessary is to replace the dot with a `\vrule` in macro `\point` as shown below.

```
\def\vr{\vrule height.1pt width.1pt}
\def\point#1#2{% keep this percent sign!
  \vbox to0pt{\kern-#2
    \hbox to0pt{\kern#1\vr\hss}\vss}%
  \ifvmode\nointerlineskip\fi}
  \fi
  }
```

Answers to Exercises

1. Zero, since the depth of a dot is zero. This is easy to verify by `\setbox0=\hbox{.}`, `\showthe\dp0`

2. Generalizing Bresenham's method for ellipses is straightforward and produces:

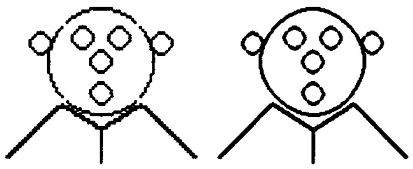
```
$x := 0$; $y := a$;
$D := (a/b)^2$; $d := 2D + 1 - 2a$;
**while** $x < y$ **do**
plot8($x, y$);
**if** $d < 0$
  $d := d + D(4x + 6)$;
**else**
  $d := d + 4D(x - y) + 6D + 4$;
  $y := y - 1$;
$x := x + 1$;
**end while**;
**if** $x = y$ **then** plot8($x, y$);
**end**;
```

References

1. Hendrickson, A., *Some Diagonal Line Hacks*, TUGboat **6**(2)83–86, (July 1985).
2. Cameron, A. G. W., *Wiggly lines*, TUGboat **6**(3)155–156, (Nov. 1985).
3. Artwick, B., *Computer Graphics*, Prentice-Hall, Englewood Cliffs, NJ.: 1985.
4. Bresenham, J. E., *A Linear Algorithm for Incremental Display of Circular Arcs*, Comm. ACM **20**(2)100–106(Feb. 1977).
5. Knuth, D. E., *The TeXbook*, Addison-Wesley, Reading, MA.: 1983.
6. Blinn, J. F., *How Many Ways Can You Draw a Circle?*, IEEE Comp. Graphics & Applic. **7**(8)39–44(Aug. 1987).
7. Hearn, D., & J. P. Baker, *Computer Graphics*, Prentice-Hall, Englewood Cliffs, NJ.: 1986.
8. Rogers, D. F., & J. A. Adams, *Mathematical Elements for Computer Graphics*, 2nd ed., McGraw-Hill, New York, NY.: 1989.
9. Smith, L. B., *Drawing Ellipses with a Fixed Number of Points*, The Computer J., **14**(1)81–86, Feb. 1971.

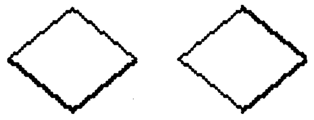
Tests

Question: What is the difference between these two identical twins?



Answer: The one on the left was done with `\step=1pt`; the one on the right, with `\step=.5pt`

Compare the two diamonds. It is easy to tell which parts are done with quadrantal and which, with octantal DDA.



◇ David Salomon
California State University, Northridge
Computer Science Department
School of Engineering and Computer Science
18111 Nordhoff Street
Northridge, CA 91330
`bccscdxs@csunb.csun.edu`

Editor's note: The methods described in this article might be applicable to a graphics system of the kind sought by David Rogers in his challenge of TUGboat 10#1 (p. 39).

Editor's note: In these macros, the names used by several `plain` control sequences (`\b`, `\B`, `\c`, `\d`, `\S`, `\t`, `\u`) have been reassigned with `\newdimen`. Beware that these names will remain associated with dimension registers even if `\begingroup` ... `\endgroup` is used in an attempt to localize their effects.