A Rasterizing Algorithm for Drawing Curves

Multimedia und Softwareentwicklung

Technikum-Wien

Alois Zingl

Wien, 2012

Abstract

This work deals with rasterizing of curves. This process converts continues geometric curves of the vector format into images of discrete pixels. Rasterizing is a fundamental task in computer graphics. The problem is illustrated on lines based on established methods. A common method is developed on the basis of the implicit equation of the curve. This principle is then proven on circles and ellipses. Subsequently the algorithm is applied on more complex curves like Béziers. This algorithm selects the closest mesh pixel to the true curves and makes it possible to draw complex curves nearly as quickly and easily as simple lines. The thereby emerging problems are considered and various solutions outlined. The method is then applied on quadratic and cubic Beziers of non-rational and rational forms as well as splines. Finally a common algorithm is established to draw any curve.

Keywords: grafic, rasterizing, curves, algorithms, Bézier, anti-aliasing, spline

Kurzfassung

Diese Arbeit beschäftigt sich mit der Rasterung von Kurven. Dabei werden kontinuierliche geometrische Kurven vom Vektorformat in Bildern aus diskreten Pixel umgewandelt. Rasterung ist eine grundsätzliche Aufgabe in der Computergrafik. Das Problem wird, ausgehend von etablierten Verfahren, an Linien erläutert. Danach wird ein allgemeines Verfahren anhand der impliziten Gleichung der Kurve erarbeitet. Dieses Prinzip wird dann an Kreisen und Ellipsen erprobt. Anschließend wird der Algorithmus an komplexeren Kurven wie Béziers ausgearbeitet. Der Algorithmus wählt jenes Pixel, welches der Kurve am nächsten liegt und ermöglicht es komplexe Kurven fast so einfach und schnell zu zeichnen wie einfache Geraden. Die dabei entstehenden Probleme werden erörtert und unterschiedliche Lösungen entworfen. Das Verfahren wird danach an quadratischen und kubischen Béziers in nicht-rationaler und rationaler Form, sowie zum Zeichnen von Splines angewendet. Schließlich wird ein allgemeiner Algorithmus für beliebige Kurvenformen aufgestellt.

Schlagwörter: Grafik, Rasterung, Kurven, Algorithmus, Bézier, Anti-aliasing, Spline

Content

1 Introduction

Vector graphics are used in computer aided geometric design. Vector graphics are based on geometrical primitives such as points, lines, circles, ellipses and Bézier curves [Foley, 1995]. However, to be useful every curve needs to be rasterized one time on displays, printers, plotters, machines, etc. About fifty years ago J. E. Bresenham of the IBM laboratories developed an algorithm for plotters to rasterize lines [Bresenham 1965]. The problem was that processors at that time had neither instructions for multiplications or divisions nor floating point arithmetic. He found an algorithm to rasterize a line on a uniform grid of pixel using integer addition and subtraction. Later on he extended this algorithm for circles.

The algorithm of this document improves Bresenham's line algorithm and expands it for ellipses and Bézier curves.

Features of the rasterising algorithm:

- Generality: This algorithm plots lines, circles, ellipses, Bézier curves, etc.
- Efficiency: Plots complex curves close to the speed of drawing lines.
- Simplicity: The pixel loop is only based on integer additions.
- Precision: No approximation of the curve.
- Smoothness: Anti-aliasing of the curves
- Flexibility: Adjustable line thickness

The principle of the algorithm could be used to rasterize any curve.

Chapter one gives an introduction to the drawing algorithms. A common drawing algorithm is introduced and applied on lines. In chapter two the algorithm is worked out on circles and lines. Chapter three uses the algorithm on quadratic Bézier curves and explains problems that appear. Different solutions are worked out which are also applied on rational quadratic Béziers in chapter four. Chapter five examines the cubic Bézier curve and develops a drawing algorithm. The rational cubic Béziers in chapter six are plotted by an approximation. Chapter seven applies the developed algorithm to draw splines. The work concludes with a compilation of the algorithm and possible implications are explained.

All curve algorithms also contain an example implementation so that everyone can test the algorithm immediately.

1.1 Curve drawing

At first several definitions and differentiations which are used throughout this work will be given.

For example, several representations are possible to define a planar curve of two dimensions. Certain definitions are better suited to drawing algorithms than others.

Explicit curve function:

The explicit equation defines one variable as a function of another $y = f(x)$. This representation tends to be unsuitable for rasterizing since it is possible that a function may have more than one y-value for a certain x-value. The circle is an example for such a curve. The function $y=\sqrt{(r^2-x^2)}$ only defines the upper half of the circle. The whole circle therefore needs two function definitions.

Implicit curve function:

The implicit equation of a curve is the zero set of a function of two variables $f(x,y) = 0$. The algebraic curves considered in this work can be represented by bivariate polynomials of real coefficients: $f(x,y)=\sum_i \sum_j a_{ij} x^i y^j = 0 \quad [i+j \le n]$.

Every point (x, y) on the curve fulfills this equation.

The maximum value of n of the equation defines the degree of the implicit function.

Parametric curve function:

The parametric equation of a curve is a vector valued function of a single variable. Points on the curve are defined by the values of the two functions $x = f_x(t)$ and $y = f_y(t)$ at the parameter values of t. A restricted interval of the parameter t defines a limited curves segment.

Certain curves like Béziers can be more easily defined by parametric representation than by others. It also enables a quick computation of the (x, y) coordinates on the curve for drawing purposes.

Gradient of curves:

The slope or gradient of a curve at point (x, y) is defined as the first derivative of the function: dy/dx. A drawing algorithm may rely on a continuously rising or falling curve. It may therefore be necessary to subdivide a curve where the drawing direction changes. These stationary points are the maximum and minimum on the curve where the slope of the curve is horizontal or vertical. These points could be calculated by setting the derivative of the function to zero in the x- or y-direction.

Certain algorithms also need different procedures for slopes below or above a value of one. Since if the gradient is below one the x-step always happens and a conditional y-step is necessary. If the slope is above one then the y-step is used and a conditional x-step is necessary.

Vector graphic versus pixel image:

The visual world of electronic multimedia consists of two opposite areas: image processing and computer graphics. [Foley, 1995]

The one side brings images of the real world into the computer. Still or movie cameras are used to make images which could be processed by computers. The other side creates artificial images inside the computer and brings them into the real world. These images are made with the help of computer aided design (CAD). [Foley, 1995]

Real world images consist of a two dimensional matrix of picture elements (pixels). Each pixel holds the information of the color at that specific position. Information which is not captured at the time of the record is lost forever. It is for example not possible to increase the resolution of an image later on to increase the details.

Artificial images generated by the computer mainly use geometric primitives such as points, lines, areas, volumes etc. Vector graphic holds the information of the position in two or three dimensions plus attributes like line color, thickness, type, etc. This information does not depend on a certain resolution. [Foley, 1995]

Fonts are a good example for vector graphics. Regardless of where you read this work the letters of this text consist of vector graphic and had to be rasterized to pixel images so you can read the text.

1.2 Rasterizing

Vector graphics are only numbers handled by the computer. To visualize vector graphics they must be digitized into a grid of pixel. This conversion is called rasterizing. Whereas the conversion of pixel images to vector graphics is difficult, the other way is comparatively simple. That is a benefit since rasterizing is needed every time to make the numbers visible. Rasterizing is required for all output devices like monitors, beamers, printers, plotters, etc. Computational efficiency is therefore an important goal of this work. [Foley, 1995]

It is not possible to mention all works related to rasterizing. A few of the recent publications together with main ideas follow as an inspiration for possible algorithms.

1.2.1 Related work

Foley describes two ways to draw a parametric curve [Foley, 1995]. The first is by iterative evaluation of $f_x(t)$ and $f_y(t)$ for incrementally spaced values of t. The second is by recursive subdivision until the control points get sufficiently close to the curve. Both methods have their benefits and disadvantages. This document describes a third way by transforming the parametric equation of the curve into the implicit equation and drawing the curve by iterative evaluation of the implicit equation.

Let's start with a simple line. How can a line from P_0 to P_1 be rasterized? Going through all x-positions of the pixels of the line, the y-positions can be calculated by

$$y = (x-x_0)(y_1-y_0)/(x_1-x_0)+y_0.$$

This method has a drawback. It needs floating point multiplication and division. That may not seem to be difficult to calculate. But is it possible to do it more efficiently?

The expression to calculate the y position contains the ratio Δx/Δy of the slope. Instead of a fraction it is possible to make the calculation by the integer numbers of numerator and denominator. This solution avoids floating pointing calculations.

Every x-step the y difference is added. If the expression is larger than the x-difference this difference is subtracted and a y-step is made. This algorithm is called Bresenham algorithm. [Bresenham, 1965]

But this solution only works if the y-difference is smaller than the x-difference. For the other case a second procedure with exchanged coordinates is necessary. This algorithm steps through all y-positions and calculates the corresponding x-positions. The need of two different procedures for the same algorithm is a handicap for simplification and extension to use it for more complex curves.

[Loop et al., 2005] presents a resolution independent drawing algorithm which uses programmable graphics hardware to perform the rendering. This approach has the advantage that anti-aliasing could also be calculated by the graphical processor .

The algorithm of this document focuses on curves up to the polynomial degree of three. Higher polynomial degrees contain multiple singular points or close curve segments which cannot be handled by the algorithm and need special solutions. Such curves could be drawn by algorithms of sub-pixeling worked out by [Emeliyanenko, 2007] or distance approximations in the work of [Taubin, 1994].

This solution of the drawing algorithm is similar to the work of [Golipour-Koujali, 2005], Yang [Yang et al., 2000] and Kumar [Kumar et al., 2011]. The main difference to their work is that instead of eight different procedures for every octant of the drawing direction, a common algorithm for any direction is developed. This makes the algorithm more compact.

1.2.2 Midpoint algorithm

The line in Figure 1 should be drawn. Pixel P is already set. For simplification the slope of the line is assumed to be between zero and one. Under these conditions only two pixels are possible to be set next: P_X or P_{XY}. The decision which one should be set could be made by the distance of the line to the pixel center. If the line is closer to Pixel P_X then this pixel is set. Otherwise pixel P_{XY} is set. Instead of the distance to the line the implicit function of the curve is used $f(x,y)=0$. The function is zero for points on the line. It is positive for points at the upper left side and negative for points at the lower right side. The criteria could therefore be made by the value of the implicit function at the point P_M exactly between P_X and P_{XY}. If the function on Point P_M is positive point P_{XY} is set, if it is negative point P_X is set. Since the value of the midpoint between the two pixels is defined as the decision this algorithm is called midpoint algorithm. [Foley, 1995]

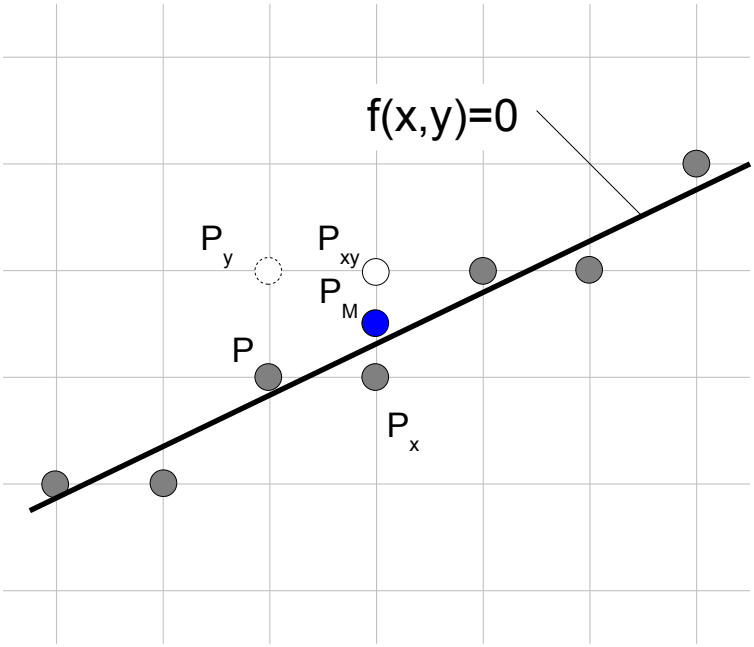


Figure 1: Midpoint algorithm

This method is limited to slopes between zero and one. In case of the line a similar algorithm is needed for slopes above one to decide between the points P_{XY} and P_Y.

1.2.3 Horner's algorithm

Another way of drawing complex curves is forward differencing by Horner's algorithm [Foley, 1995]. This algorithm calculates the value of a function just by adding the difference to the previous value: $f(t+\Delta t) = f(t)+d$. If the function is a polynomial of degree one $f(t) = a_1 t+a_0$ the difference is only a constant value: $d_1 = \Delta t\, a_1$. For degree n polynomials the differences make successive additions: $d_i = d_i+d_{i+1}$. The initialization values of d_i could be calculated by the differences of the function $f(t)$. If this algorithm is applied on the parametric equation of the curve $x = f_x(t)$ and $y = f_y(t)$ the coordinates of Béziers for example could be calculated only by additions. The problem with this algorithm is to choose an appropriate step size Δt. If this step is too large a few pixels are omitted and if it is too small the same pixel is set multiple times.

Horner's algorithm is not limited to lines. It could be used for other curves too. The implicit function of the curve is needed. Starting at position P every octant of the drawing direction needs a decision if the pixel in one of the eight appropriate direction should be set or not. This document somehow applies Horner's algorithm on the implicit equation of the curve.

Another way of rastering a curve is approximation. The curve is subdivided into short lines and each line is plotted separately. But approximation also means to choose one of two disadvantages. If the approximation should be accurate the curve must be divided in many small segments. This is computationally expensive. On the other hand the curve becomes

edgy if the approximation is not accurate enough. A fast and accurate rasterizing algorithm for curves is therefore desirable.

1.3 Problem definition

The implicit equation $f(x,y) = 0$ defines a curve from the point $P_0(x_0,y_0)$ to $P_1(x_1,y_1)$. The gradient of the curve must continuously be either positive or negative. This restriction is solved by subdividing of the curve.

The curve could be a straight line, but also be part of an ellipse or a Bézier curve for example.

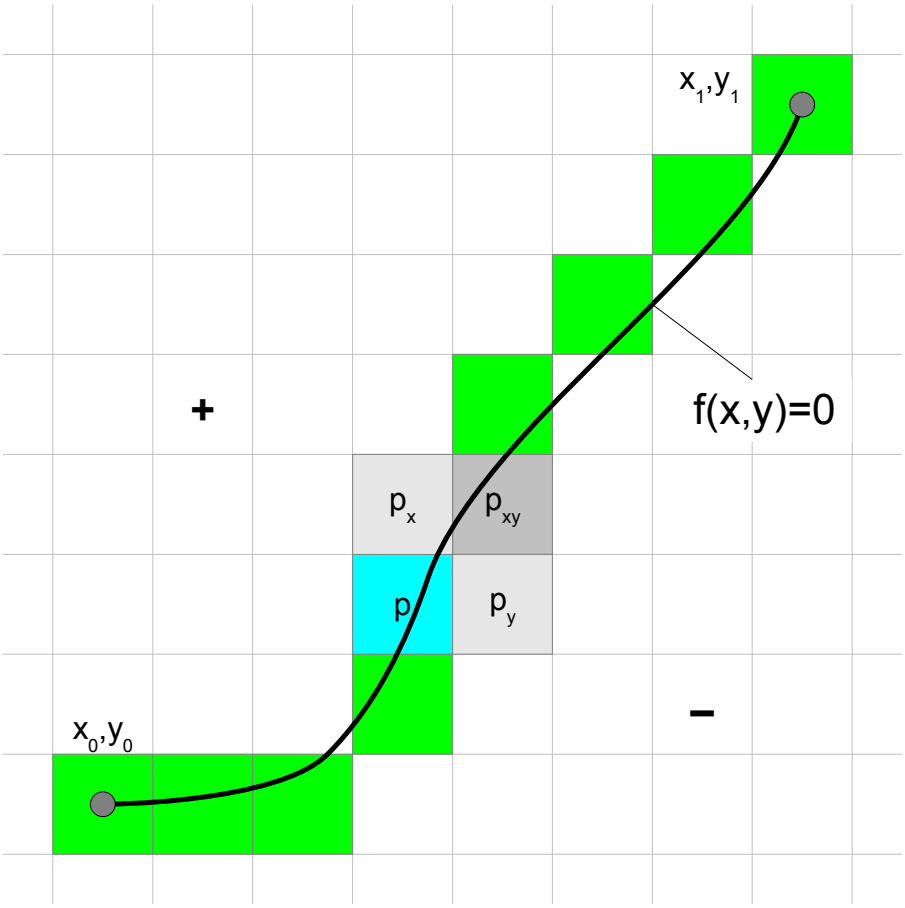


Figure 2: Pixel grid of curve f(x,y)=0

The curve in figure 2 should be digitized into a grid of pixel. This conversion is called rasterizing.

Which pixel on the grid should be set next to represent the curve in figure 2 most suitably?

1.4 General solution

An error *e* of the pixel p is introduced by the algorithm as a measurement for the deviation of the pixel from the curve: *e* = f(*x*,*y*). The error value is zero for pixels exactly at the curve, positive for one side and negative for the other side of the curve. This error calculation is

used for the decision which pixel should be set next. Starting by pixel p only between three possible pixel could be chosen for the next pixel because of the positive gradient: P_X or P_{XY}.

The algorithm starts with the assumption that point P_{XY} will be set next. So if the error $|e_{xy}|$ of point P_{XY} is lesser than the error $|e_x|$ of pixel P_X than the x direction will be incremented. The same decision is considered for the y direction. If the error $|e_{xy}|$ of P_{XY} is lesser than the error $|e_y|$ than the y direction will be incremented. That's why in figure 2 the pixel above P is labeled P_X and besides P is labeled P_Y.

Since a positive gradient is assumed and the error on one side of the curve will be negative, the unequation $e_x \geq e_{xy} \geq e_y$ will always be true which makes it possible to avoid the calculation of the absolute value for the comparison. The conditions for the increments are now:

$$\text{if } e_x + e_{xy} > 0 \text{ then increment x}$$
$$\text{if } e_y + e_{xy} < 0 \text{ then increment y}$$

The benefit of this approach is that the error of the present pixel is already known, so only the difference to the previous pixel has to be calculated. And this computation is more efficient to implement than calculating the entire expression for every pixel.

The error of the next pixel has to be calculated for all three possibilities of the actual pixel P: e_x, e_y and e_{xy}. Could that be further reduced? If the algorithm doesn't track the error e of the *current* pixel p but the error e_{xy} of the *next diagonal* pixel P_{XY} then only two error calculations had to be done: e_x and e_y. Because the error e is not available e_x and e_y must be calculated as one pixel *less* from the actual error.

1.5 Pseudo code of the algorithm

The calculation of the error value depends on the curve function but the condition for the increment will always be the same.

```
set up x, y to x0, y0
set up error variable exy for P(x0+1,y0+1)
loop
    set pixel x, y
    if ex + exy > 0 then increment x, sub difference error
    if ey + exy < 0 then increment y, add difference error
loop until end pixel
```

Listing 1: Pseudo code of the algorithm

Please note that if the condition is true the difference error must be calculated after the increment is made since the error calculation always looks one diagonal pixel ahead.

A few algorithms in this document contain many details. Not all are explicitly mentioned in the text. Certain minor implementation solutions could be better and more concisely explained by sample code. The programming language C is used since it could be easily converted to other languages. Drawing curves also is a system task and most operating systems are written in this language. The examples make it also possible to test the algorithm immediately.

The bit size of the variables is sometimes critical and is assumed to be at least 16 bit for *int*, 32 bit for *long* or *float* and 64 bit for *double*.

1.6 Straight lines

The implicit equation for a straight line from point P(x_0,y_0) to P(x_1,y_1) is:

$$(x_1-x_0)(y-y_0)-(x-x_0)(y_1-y_0) = 0 \tag{1}$$

With the definition of $dx = x_1-x_0$ and $dy = y_1-y_0$ the error *e* makes then:
$e = (y-y_0)dx-(x-x_0)dy$.

The following calculations are simple but a bit confusing because of the indexes and the signs.

The error of the diagonal step makes: $e_{xy} = (y+1-y_0)dx-(x+1-x_0)dy = e+dx-dy$.

The error calculations for the x and y directions make: $e_x = (y+1-y_0)dx-(x-x_0)dy = e_{xy}+dy$ and $e_y = (y-y_0)dx-(x+1-x_0)dy = e_{xy}-dx$.

The error for the first step makes: $e_1 = (y_0+1-y_0)dx-(x_0+1-x_0)dy = dx-dy$.

Figure 3 shows a line with *dx* = 5 and *dy* = 4. The error value of the cyan pixel is 1. The three gray pixels are possible next choices. Increase in x-direction subtracts 4 (*dy*), increase in y-direction adds 5 (*dx*) to the error value. The dark gray pixel has the lowest absolute value. It is calculated in advance since the error value is one diagonal pixel ahead.
Since $e_{xy}+e_y$ = +2–3 = –1 is less than zero the y direction is increased and *dx* added to the error value. The same is done for the other direction: $e_{xy}+e_x$ = +2+6 = +8 is greater than zero, so the x-direction is increased and *dy* subtracted from the error value.

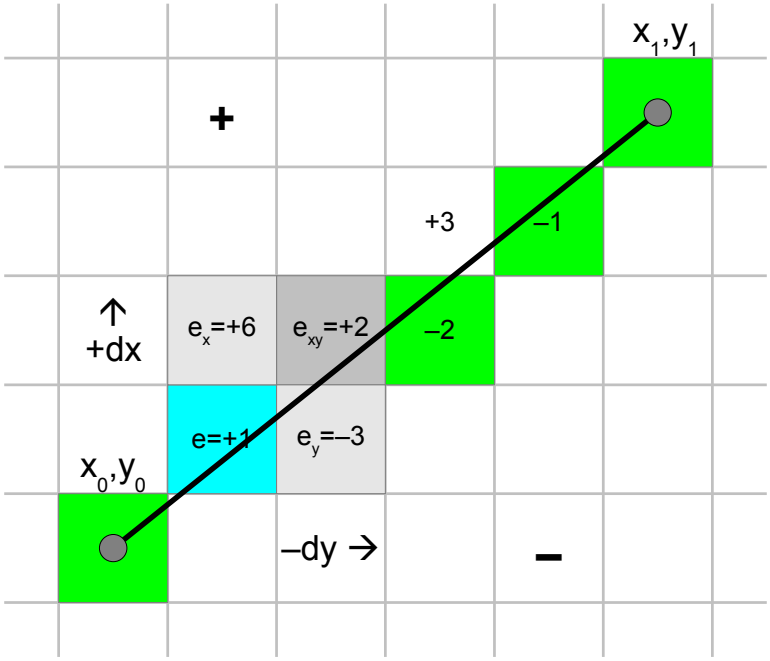


Figure 3: line with error values (dx=5, dy=4)

Although the x and y direction seemed to be interchanged in figure 3 they are actually not. Just the error increment for the test condition is interchanged.

1.7 Program to plot a line

There are different possibilities to handle negative gradients or reversed lines. The solution used here is to negate the step direction.

```
void plotLine(int x0, int y0, int x1, int y1)
{
   int dx =  abs(x1-x0), sx = x0<x1 ? 1 : -1;
   int dy = -abs(y1-y0), sy = y0<y1 ? 1 : -1;
   int err = dx+dy, e2;                                /* error value e_xy */

   for (;;){                                                    /* loop */
      setPixel(x0,y0);
      e2 = 2*err;
      if (e2 >= dy) {                                    /* e_xy+e_x > 0 */
         if (x0 == x1) break;
         err += dy; x0 += sx;
      }
      if (e2 <= dx) {                                    /* e_xy+e_y < 0 */
         if (y0 == y1) break;
         err += dx; y0 += sy;
      }
   }
}
```

Listing 2: Program to plot a line

There is no approximation by the algorithm. So the error value of the last pixel is always exactly zero. This version is optimized to check for end-of-the-loop only if the corresponding direction is incremented.

Because this algorithm works in x and y direction symmetrically it needs an additional if condition in the pixel loop, one more than the traditional Bresenham's line algorithm. It is possible to avoid this additional condition if it is known in advance that the gradient of the line is always below or above one.

The program also elegantly illustrates the xy-symmetry of Bresenham's line algorithm. The same considerations could now be applied to curves of higher polynomial degree.

Due to the symmetry of the line it is also possible to start the drawing form both ends of the line and stop in the middle. This approach may speed up the drawing but introduces slight irregularities in the line.

2 Ellipses

Wouldn't it be easier to start with the symmetric circle instead of the more complicated expression of the ellipse?

It would be, but only a bit. The calculations for ellipses are not so difficult so the solution can easily be adapted for circles by stetting $a = b = r$ and had not to be done again.

By a proper choice of the coordinate system an ellipse can be described by the implicit equation: $$x^2b^2+y^2a^2-a^2b^2 = 0 \quad (2)$$

The equation for the error calculation for the pixel is therefore: $e = x^2b^2+y^2a^2-a^2b^2$.

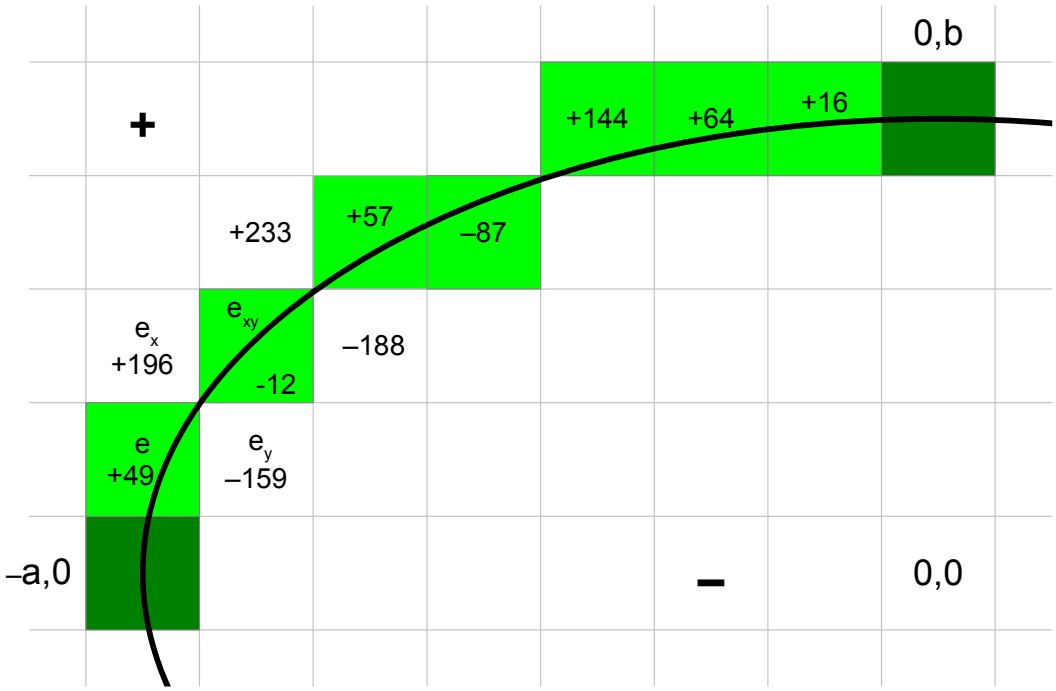


Figure 4: ellipse quadrant with error values for a=7 and b=4

The error of the next diagonal pixel $x+1, y+1$ makes:

$$e_{xy} = (x+1)^2b^2+(y+1)^2a^2-a^2b^2 = e+(2x+1)b^2+(2y+1)a^2 .$$

The error of the next pixel $x+1$ makes: $e_y = (x+1)^2b^2+y^2a^2-a^2b^2 = e_{xy}-(2y+1)a^2$ and for $y+1$:

$$e_x = x^2b^2+(y+1)^2a^2-a^2b^2 = e_{xy}-(2x+1)b^2.$$

The ellipse is subdivided into four quadrants. The second quadrant is used because of the positive gradient. It starts at pixel P(–*a*,0) and ends at P(0,*b*).

The error of the first pixel is therefore: $e_1 = (-a+1)^2b^2+(0+1)^2a^2-a^2b^2 = a^2-b^2(2a-1)$.

We are now able to build the algorithm.

2.1 Program to plot an ellipse

With these preparations it is easy to write the algorithm down. But the ellipse needs special treatment if it is very flat.

```
void plotEllipse(int xm, int ym, int a, int b)
{
   int x = -a, y = 0;   /* II. quadrant from bottom left to top right */
   long e2 = (long)b*b, err = x*(2*e2+x)+e2;       /* error of 1.step */

   do {
       setPixel(xm-x, ym+y);                          /*   I. Quadrant */
       setPixel(xm+x, ym+y);                          /*  II. Quadrant */
       setPixel(xm+x, ym-y);                          /* III. Quadrant */
       setPixel(xm-x, ym-y);                          /*  IV. Quadrant */
       e2 = 2*err;
       if (e2 >= (x*2+1)*(long)b*b)                    /* e_xy+e_x > 0 */
          err += (++x*2+1)*(long)b*b;
       if (e2 <= (y*2+1)*(long)a*a)                    /* e_xy+e_y < 0 */
          err += (++y*2+1)*(long)a*a;
   } while (x <= 0);

   while (y++ < b) {           /* to early stop of flat ellipses a=1, */
       setPixel(xm, ym+y);                 /* -> finish tip of ellipse */
       setPixel(xm, ym-y);
   }
}
```

Listing 3: Simple program to plot an ellipse

The algorithm stops too early when the radius *a* of the ellipse equals one. In such cases the strategy of looking ahead fails because it tests the pixel of the adjacent quadrant at the end. On normal condition this doesn't matter since the ellipse is already finished. But for *a* = 1 the algorithm has to finish the tip of the ellipse by an additional loop (lines 18-21 in listing 3).

The algorithm could be concatenated to draw four consecutive quadrants of the ellipse, which is necessary for plotters. This way it is also possible to draw just a certain arc of an ellipse from angle α to β. Only the start position and the error values need to be calculated differently.

The value of the error could get huge. Its variables (and the comparison with them) must be able to hold the triple word size of the radii *a, b* to avoid an overflow. (If *a,b* have 16 bit then `err` must have 48 bit at least.)

2.2 Optimized program to plot an ellipse

The algorithm could be further optimized for speed by introducing two additional increment variables.

```
void plotOptimizedEllipse(int xm, int ym, int a, int b)
{
   long x = -a, y = 0; /* II. quadrant from bottom left to top right */
   long e2 = b, dx = (1+2*x)*e2*e2;                /* error increment  */
   long dy = x*x, err = dx+dy;                      /* error of 1.step */

   do {
       setPixel(xm-x, ym+y);                          /*   I. Quadrant */
       setPixel(xm+x, ym+y);                          /*  II. Quadrant */
       setPixel(xm+x, ym-y);                          /* III. Quadrant */
       setPixel(xm-x, ym-y);                          /*  IV. Quadrant */
       e2 = 2*err;
       if (e2 >= dx) { x++; err += dx += 2*(long)b*b; }    /* x step */
       if (e2 <= dy) { y++; err += dy += 2*(long)a*a; }    /* y step */
   } while (x <= 0);

   while (y++ < b) {    /* to early stop for flat ellipses with a=1, */
       setPixel(xm, ym+y);                /* -> finish tip of ellipse */
       setPixel(xm, ym-y);
   }
}
```

Listing 4: Optimized program to plot an ellipse

Of course it is also suitable to introduce variables for the constants of $2b^2$ and $2a^2$.

This algorithm makes the drawing of an ellipse as easy as drawing a line: only integer addition and subtraction are necessary.

There is no approximation done by the algorithm. The error value of the last pixel will always be exactly zero.

2.3 Rasterizing circles

The previous algorithm could be changed to draw a circle by setting $a = b = r$. The calculation of the error value could be simplified by dividing it by r^2.

But some circles like that in figure 5 are looking strange with additional points. Is something wrong with the algorithm?

Only four pixels at the 45 degree diagonals are affected when x equals y.

A closer examination of the error values in figure 5 shows that the points in question have a lower absolute value than the adjacent pixel.

From the algorithm's point of view the four suspicious pixels must be set because they are closest to the circle. The problem occurs since this is the point where the gradient of the circle changes from below to above 45 degree. That is the reason why the adjacent pixels also have to be set since they have a lower error value than the alternatives. So nothing is wrong with the algorithm, just unfortunate mathematical coincidence.

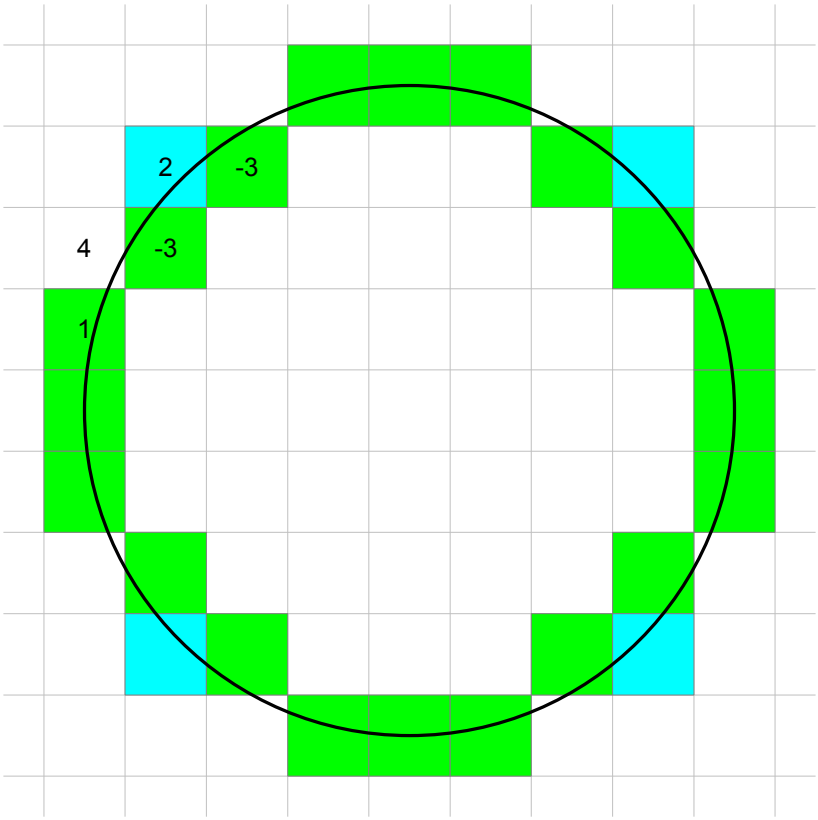


Figure 5: spurious pixel on a circle of radius 4

For which circles do the spurious pixels appear? The sequence of numbers of the radii is curious: 4, 11, 134, 373, 4552, …

According to the On-Line Encyclopedia of Integer Sequences (http://oeis.org/A055979) the rule for odd n makes: $\frac{(3+\sqrt{8})^{n+1}-(3-\sqrt{8})^{n+1}-2((3+\sqrt{8})^{n}-(3-\sqrt{8})^{n})}{2\sqrt{8}}$ and for even n: $\frac{(3+\sqrt{8})^{n+1}-(3-\sqrt{8})^{n+1}-2((3+\sqrt{8})^{n-1}-(3-\sqrt{8})^{n-1})}{6\sqrt{8}}$. (3)

Is it possible to avoid these spurious pixels although they appear rarely?

One simple way would be by adding the constant value one to the variable err at the initialization making all radii a bit smaller. But this also changes certain other circles, especially small ones look strange then.

In normal cases these additional pixel will hardly be noticed.

This problem will occur on other curves too.

One possibility to avoid the unwanted pixels is to include an additional 'spurious pixel' check when y is incremented. These additional pixel occur on y steps when no second y step is done (and no x step happens). An additional error check looking one pixel ahead avoids the spurious pixel.

```
void plotCircle(int xm, int ym, int r)
{
   int x = -r, y = 0, err = 2-2*r; /* bottom left to top right */
   do {
      setPixel(xm-x, ym+y);              /*   I. Quadrant +x +y */
      setPixel(xm-y, ym-x);              /*  II. Quadrant -x +y */
      setPixel(xm+x, ym-y);              /* III. Quadrant -x -y */
      setPixel(xm+y, ym+x);              /*  IV. Quadrant +x -y */
      r = err;
      if (r <= y) err += ++y*2+1;               /* e_xy+e_y < 0 */
      if (r > x || err > y)   /* e_xy+e_x > 0 or no 2nd y-step */
         err += ++x*2+1;                       /* -> x-step now */
   } while (x < 0);
}
```

Listing 5: Circle program to avoid spurious pixel

In contrast to the ellipse the circle algorithm also avoids setting certain pixels twice. It could be further changed to eight octants setting 8 pixels per loop and looking much like other circle algorithms.

2.4 Squaring the ellipse

Some-times an algorithm is needed to plot circles or ellipses where the corners of a surrounding rectangle are specified instead of center and radius. This would also include circles or ellipses with a diameter of odd pixels, something the algorithms before were not able to do.

The algorithm must calculate on a grid of double resolution to plot such ellipses. On this grid the algorithm always takes double steps. If the radius *b* has a fraction of ½ then the y-direction starts with an offset y_b of one, if it is an integer the offset is zero.

The error of the next diagonal pixel $x+2$, $y+2$ makes: $e_{xy} = (x+2)^2b^2+(y+2)^2a^2-a^2b^2$.

The error of the next pixel x+2 makes: $e_y = (x+2)^2b^2+y^2a^2-a^2b^2 = e_{xy}-4(y+1)a^2$ and for y+2: $e_x = x^2b^2+(y+2)^2a^2-a^2b^2 = e_{xy}-4(x+1)b^2$.

The error of the first pixel is therefore: $e_1 = (-a+2)^2b^2+(y_b+2)^2a^2-a^2b^2 = (y_b+2)^2a^2-4(a-1)b^2$.

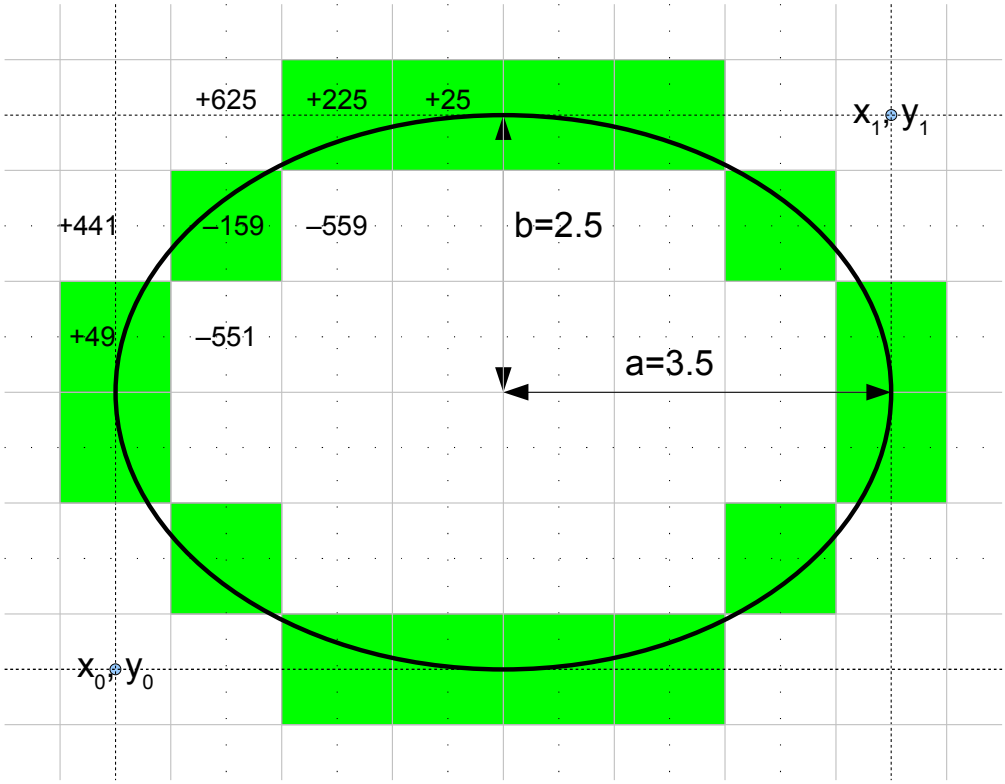


Figure 6: ellipse enclosed by a rectangle of 7x5 pixel

2.5 Program for an ellipse inside a rectangle

This is the optimized version to plot an ellipse inside a specified rectangle. Instead of 64-bit integer it uses floating point arithmetic to avoid an overflow in the error calculation. It also uses an additional check of the x-step to avoid spurious pixels.

```
void plotEllipseRect(int x0, int y0, int x1, int y1)
{                       /* rectangular parameter enclosing the ellipse */
   long a = abs(x1-x0), b = abs(y1-y0), b1 = b&1;          /* diameter */
   double dx = 4*(1.0-a)*b*b, dy = 4*(b1+1)*a*a;    /* error increment */
   double err = dx+dy+b1*a*a, e2;                    /* error of 1.step */

   if (x0 > x1) { x0 = x1; x1 += a; } /* if called with swapped points */
   if (y0 > y1) y0 = y1;                            /* .. exchange them */
   y0 += (b+1)/2; y1 = y0-b1;                         /* starting pixel */
   a = 8*a*a; b1 = 8*b*b;

   do {
      setPixel(x1, y0);                               /*   I. Quadrant */
      setPixel(x0, y0);                               /*  II. Quadrant */
      setPixel(x0, y1);                               /* III. Quadrant */
      setPixel(x1, y1);                               /*  IV. Quadrant */
      e2 = 2*err;
      if (e2 <= dy) { y0++; y1--; err += dy += a; }          /* y step */
      if (e2 >= dx || 2*err > dy) { x0++; x1--; err += dx += b1;} /* x */
   } while (x0 <= x1);

   while (y0-y1 <= b) {          /* to early stop of flat ellipses a=1 */
      setPixel(x0-1, y0);                    /* -> finish tip of ellipse */
      setPixel(x1+1, y0++);
      setPixel(x0-1, y1);
      setPixel(x1+1, y1--);
   }
}
```

Listing 6: Program to plot an ellipse enclosed by a rectangle

This algorithm works for all values of x0, y0, x1 and y1.

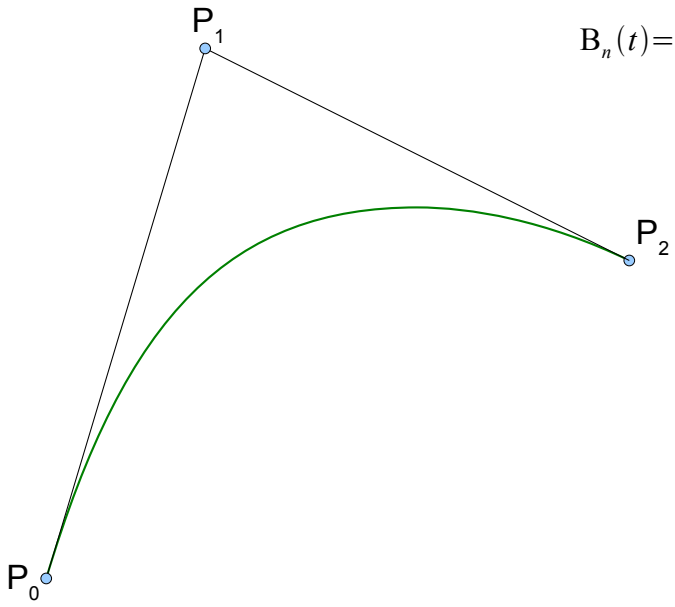
The algorithm for r otated ellipses is developed later since the direct drawing algorithm runs into troubles but could be implemented by using rational Béziers.

3 Quadratic Bézier curves

The concept of universal curves were independently developed by the French engineers Pierre Étienne Bezier from Renault and Paul de Faget de Casteljau from Citroën at the advent of the computer aided manufacturing in the car industry to design automobile bodies. [Bézier, 1986] [Casteljau, 1963]

Bézier curves consist of a set of control points. The number of points define the order of the curve.

The general Bézier equation of order *n* in parametric form given *n* +1 points P_i is defined to be [Marsh, 2005, p. 135]



$$B_n(t)=\sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^i P_i \quad (4)$$

Figure 7: Bézier curve of degree 2

This is a straight line for order *n* = 1. For order *n* = 2 this is the quadratic Bézier curve

$$B_2(t)=(1-t)^2 P_0+2(1-t)t P_1+t^2 P_2. \quad (5)$$

The conical implicit equation of the Bézier curve is needed for the algorithm.
The general implicit equation of degree 2 makes:

$$A x^2+2Bxy+Cy^2+2Dx+2Ey+F=0.$$

This equation has six unknown coefficients so six linearly independent equations are needed to derive the unknowns. If F is non zero the equation could be divided by F:

$a x^2+2bxy+cy^2+2dx+2ey+1=0$ leaving five unknowns.

Two could be derived by setting $x=x_0 \rightarrow y=y_0$ and $x=x_2 \rightarrow y=y_2$:

$a x_0^2+2b x_0 y_0+c y_0^2+2d x_0+2e y_0=-1$ and $a x_2^2+2b x_2 y_2+c y_2^2+2d x_2+2e y_2=-1.$

A third could be derived from the parametric form by setting $t = ½$:

$$B_2\left(\frac{1}{2}\right)=\frac{P_0+2P_1+P_2}{4}$$

$$a(x_0+2x_1+x_2)^2+2b(x_0+2x_1+x_2)(y_0+2y_1+y_2)+c(y_0+2y_1+y_2)^2+... \quad (6)$$
$$...+8d(x_0+2x_1+x_2)+8e(y_0+2y_1+y_2)=-16$$

The last two unknowns are computed by the derivative of the implicit equation:

$$\nabla_{x,y}=\langle 2ax+2by+2d, 2bx+2cy+2e\rangle$$

By the gradients at the two points P_0: $\frac{x_0-x_1}{y_0-y_1}=\frac{ax_0+by_0+d}{bx_0+cy_0+e}$ and P_2:

$\frac{x_2-x_1}{y_2-y_1}=\frac{ax_2+by_2+d}{bx_2+cy_2+e}$ the equations for the last two unknowns could be derived:

$$ax_0(x_0-x_1)+b(y_0(x_0-x_1)+x_0(y_0-y_1))+cy_0(y_0-y_1)+d(x_0-x_1)+e(y_0-y_1)=0$$
$$ax_2(x_2-x_1)+b(y_2(x_2-x_1)+x_2(y_2-y_1))+cy_2(y_2-y_1)+d(x_2-x_1)+e(y_2-y_1)=0$$

The computations of the unknowns get a bit difficult now. Could they be simplified? By the substitution of $\hat{P}_i=P_i-P_1$ the Bézier curve is shifted by the offset of $-P_1$. It is no problem for the algorithm to shift it back later. So for the simplification of this computation point P_1 is assumed to be at the origin: $\hat{x}_1=\hat{y}_1=0$ and the values of x_1 and y_1 are subtracted from the other points: $\hat{x}_i=x_i-x_1$. The system of five linear equations could now be written as the matrix equation

$$\begin{bmatrix} \hat{x}_0^2 & 2\hat{x}_0\hat{y}_0 & \hat{y}_0^2 & 2\hat{x}_0 & 2\hat{y}_0 \\ \hat{x}_2^2 & 2\hat{x}_2\hat{y}_2 & \hat{y}_2^2 & 2\hat{x}_2 & 2\hat{y}_2 \\ (\hat{x}_0+\hat{x}_2)^2 & 2(\hat{x}_0+\hat{x}_2)(\hat{y}_0+\hat{y}_2) & (\hat{y}_0+\hat{y}_2)^2 & 8(\hat{x}_0+\hat{x}_2) & 8(\hat{y}_0+\hat{y}_2) \\ \hat{x}_0^2 & 2\hat{x}_0\hat{y}_0 & \hat{y}_0^2 & \hat{x}_0 & \hat{y}_0 \\ \hat{x}_2^2 & 2\hat{x}_2\hat{y}_2 & \hat{y}_2^2 & \hat{x}_2 & \hat{y}_2 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -16 \\ 0 \\ 0 \end{bmatrix} \quad (7)$$

This matrix equation can be solved like below:

$$A=(\hat{y}_0+\hat{y}_2)^2, B=-(\hat{x}_0+\hat{x}_2)(\hat{y}_0+\hat{y}_2), C=(\hat{x}_0+\hat{x}_2)^2,$$
$$D=(\hat{y}_0-\hat{y}_2)(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0), E=-(\hat{x}_0-\hat{x}_2)(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0), F=(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0)^2.$$

The implicit equation of the quadratic Bézier curve for $\hat{x}_1=\hat{y}_1=0$ makes:

$$x^2(\hat{y}_0+\hat{y}_2)^2-2xy(\hat{x}_0+\hat{x}_2)(\hat{y}_0+\hat{y}_2)+y^2(\hat{y}_0+\hat{y}_2)^2+$$
$$+2(x(\hat{y}_0-\hat{y}_2)-y(\hat{x}_0-\hat{x}_2))(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0)+(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0)^2=0.$$

The overall curvature of the Bézier curve is defined by

$$cur=\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0=(x_0-x_1)(y_2-y_1)-(x_2-x_1)(y_0-y_1). \quad (8)$$

The previous substitutions could be added again. By some computations the implicit equation of the quadratic Bézier curve is simplified to:

$$(x(y_0-2y_1+y_2)-y(x_0-2x_1+x_2))^2+2(x(y_0-y_2)-y(x_0-x_2))cur+cur^2=0. \quad (9)$$

The quadratic Bézier curve is part of a parabola.

3.1 Error calculation

The plotting algorithm relies on a continuously positive (or negative) gradient of the curve (the slope is either rising or falling). Since point $\hat{P}_1$ is the point at the origin $\hat{P}_0$ must be in the third quadrant and $\hat{P}_2$ in the first quadrant for a positive gradient. The following conditions are always true then: $\hat{x}_1=\hat{y}_1=0,\ \hat{x}_0\le 0\le\hat{x}_2$ and $\hat{y}_0\le 0\le\hat{y}_2$.

It is no problem to fulfill this requirement since the Bézier curve could be subdivided at the point where the sign of the gradient is changing, drawing two curves one after another.

The equation for the error calculation of the Bézier curve makes:

$$e(x,y)=(x(\hat{y}_0+\hat{y}_2)-y(\hat{x}_0+\hat{x}_2)+(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0))^2-4(x\hat{y}_2-y\hat{x}_2)(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0)$$

which is equivalent to

$$e(x,y)=(x(\hat{y}_0+\hat{y}_2)-y(\hat{x}_0+\hat{x}_2)-(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0))^2+4(x\hat{y}_0-y\hat{x}_0)(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0).$$

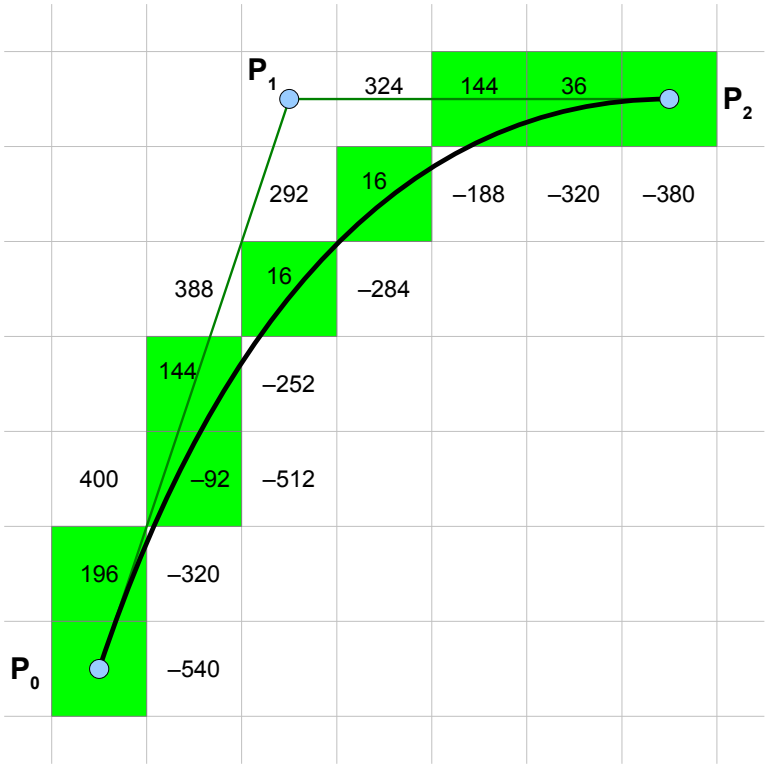


Figure 8: Error values of a quadratic Bézier curve

Only the changing terms of the step increment are of interest for the algorithm because the other terms remain constant during the pixel loop.

The initial values of these changing terms are the differences of the implicit equation and computed as follows:

$$d_x = \mathrm{e}(\hat{x} \pm 1, \hat{y}) - \mathrm{e}(\hat{x}, \hat{y}) = (1 \pm 2\hat{x})(\hat{y}_0 + \hat{y}_2)^2 \mp 2\hat{y}(\hat{x}_0 + \hat{x}_2)(\hat{y}_0 + \hat{y}_2) \pm 2\,cur(\hat{y}_0 - \hat{y}_2),$$
$$d_y = \mathrm{e}(\hat{x}, \hat{y} \pm 1) - \mathrm{e}(\hat{x}, \hat{y}) = (1 \pm 2\hat{y})(\hat{x}_0 + \hat{x}_2)^2 \mp 2\hat{x}(\hat{x}_0 + \hat{x}_2)(\hat{y}_0 + \hat{y}_2) \mp 2\,cur(\hat{x}_0 - \hat{x}_2).$$

Since this Bézier curve is of second degree the increment error changes each step too. Not only the error of the calculation has to be incremented according to the steps, but also the increment of d_x and d_y itself changes each step. In case of a quadratic polynomial this could also be computed by the second derivative.

For the step in x-direction the increment d_x is increased about

$$d_{xx} = \mathrm{e}(x+2, y) - 2\,\mathrm{e}(x+1, y) = \frac{\partial^2 \mathrm{e}}{\partial x^2} = 2(\hat{y}_0 + \hat{y}_2)^2 = 2(y_0 - 2y_1 + y_2)^2$$ and d_y is increased about

$$d_{xy} = \mathrm{e}(x+1, y+1) - \mathrm{e}(x+1, y) - \mathrm{e}(x, y+1) = \frac{\partial^2 \mathrm{e}}{\partial x \partial y} = -2(\hat{x}_0 + \hat{x}_2)(\hat{y}_0 + \hat{y}_2) = -2(x_0 - 2x_1 + x_2)(y_0 - 2y_1 + y_2).$$

For the step in y-direction the increment d_y is increased about

$$d_{yy} = \mathrm{e}(x, y+2) - 2\,\mathrm{e}(x, y+1) = \frac{\partial^2 \mathrm{e}}{\partial y^2} = 2(\hat{x}_0 + \hat{x}_2)^2 = 2(x_0 - 2x_1 + x_2)^2$$

and d_x is increased about $\frac{\partial^2 \mathrm{e}}{\partial x \partial y}$.

These increments are independent of x and y.

3.2 Troubles with slightly curved lines

So far the algorithm seems to work out nicely. But it fails when the Bézier curve becomes nearly a straight line. What happens becomes clear if the entire curve is analyzed, not only the short part the algorithm wants to plot. The curve is a symmetric parabola. It has a second part. For curves with large curvature the second half is far away, leaving a clear path the algorithm can follow. But on nearly straight lines this second half can fall within the current possible pixel! Then the algorithm is confused since it relies on a clear gradient of error values.

This problem occurred before. On flat ellipses with *a* = 1 the algorithm stopped to early. But the situation was lucky. The ellipses were always placed in symmetric orthogonal orientation. The algorithm failed only in one case which could be fixed by an extra loop.

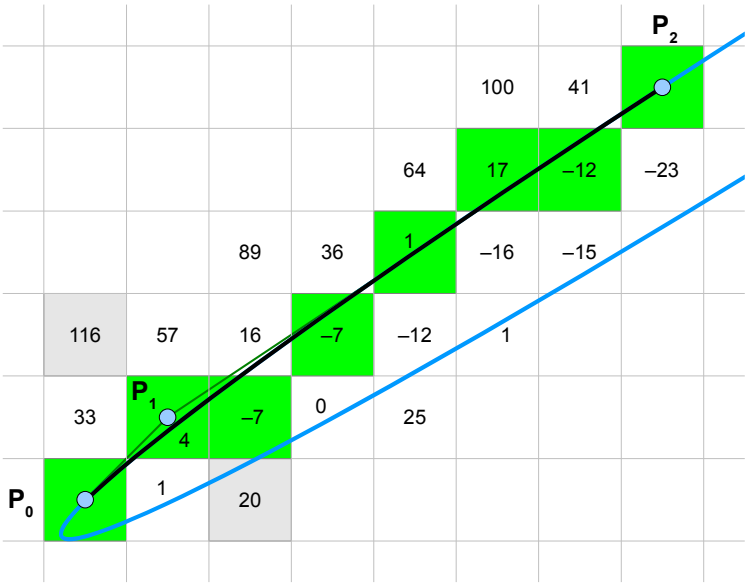


Figure 9: Algorithm in trouble: no path to follow

Figure 9 shows an ambiguous situation. No positive/negative gradient is visible to follow from point P_0. Even if the algorithm could somehow detect a way of low error values the green pixel near P_1 of value –7 would be wrongly selected since it has its low absolute value from the wrong blue half of the curve not the wanted black one.

The algorithm fails if another part of the (invisible) function comes near the vicinity of the set pixel.

What to do? The problem only occurs on almost straight lines. The Bézier points are also highly asymmetric. There are a few possibilities. One solution is to check if the error value a few pixel from the starting point still is below (x-direction) or above (y-direction) zero. When the algorithm looks n pixels ahead along the x- or y-direction from P_0 it has to check if $n(d_x+d_{xy}+(n-1)d_{yy}/2)<0$ and $n(d_y+d_{xy}+(n-1)d_{xx}/2)>0.$ But this check has also to be done for P_2 since the situation could be reversed and the end point P_2 cause the confusion. Another possibility would be to increase the resolution of the pixel raster just to find the correct pixel path of the curve and set the appropriate pixel closest to it.

This solution is not really satisfying at all. A few nearly diagonally straight lines still get additional pixels although the algorithm works. And also the question remains what to do if this algorithm fails due do this theoretical flaw? The simplest solution is to plot the curve in two or more straight lines instead.

3.3 Program to plot simple Bézier curves

The program in listing 7 accepts only basic Bézier curves without sign change of the gradient (no horizontal or vertical turns). A change of the sign would require additional computations in the pixel loop. This problem is solved later by subdivision.

```
void plotBasicQuadBezier(int x0, int y0, int x1, int y1, int x2, int y2)
{
  int sx = x0<x2 ? 1 : -1, sy = y0<y2 ? 1 : -1; /* step direction */
  double x = x0-2*x1+x2, y = y0-2*y1+y2, xy = 2*x*y*sx*sy;
  double cur = sx*sy*(x*(y2-y0)-y*(x2-x0))/2;     /* curvature */
                                /* compute error increments of P0 */
  double dx = (1-2*abs(x0-x1))*y*y+abs(y0-y1)*xy-2*cur*abs(y0-y2);
  double dy = (1-2*abs(y0-y1))*x*x+abs(x0-x1)*xy+2*cur*abs(x0-x2);
                                /* compute error increments of P2 */
  double ex = (1-2*abs(x2-x1))*y*y+abs(y2-y1)*xy+2*cur*abs(y0-y2);
  double ey = (1-2*abs(y2-y1))*x*x+abs(x2-x1)*xy-2*cur*abs(x0-x2);

                              /* sign of gradient must not change */
  assert((x0-x1)*(x2-x1) <= 0 && (y0-y1)*(y2-y1) <= 0);

  if (cur==0) { plotLine(x0,y0,x2,y2); return; } /* straight line */

  x *= 2*x; y *= 2*y;
  if (cur < 0) {                              /* negated curvature */
    x = -x; dx = -dx; ex = -ex; xy = -xy;
    y = -y; dy = -dy; ey = -ey;
  }
  /* algorithm fails for almost straight line, check error values */
  if (dx >= -y || dy <= -x || ex <= -y || ey >= -x) {
    x1 = (x0+4*x1+x2)/6; y1 = (y0+4*y1+y2)/6;    /* approximation */
    plotLine(x0,y0, x1,y1);
    plotLine(x1,y1, x2,y2);
    return;
  }
  dx -= xy; ex = dx+dy; dy -= xy;              /* error of 1.step */

  for(;;) {                                          /* plot curve */
    setPixel(x0,y0);
    ey = 2*ex-dy;                /* save value for test of y step */
    if (2*ex >= dx) {                                   /* x step */
      if (x0 == x2) break;
      x0 += sx; dy -= xy; ex += dx += y;
    }
    if (ey <= 0) {                                      /* y step */
      if (y0 == y2) break;
      y0 += sy; dx -= xy; ex += dy += x;
    }
  }
}
```

Listing 7: Program to plot a basic Bézier curve

A few comments to listing 7:
A negative curvature negates the gradient of the error values. This leaves the possibility to either negate the other values too or use another pixel loop with interchanged condition ($e_{xy}+d_x<0 \rightarrow$ x++).

The error increments are used to look three pixel in x and y-direction ahead and detect almost straight lines by an additional gradient change of the error values. This is done for both ends at which the increments of P_2 are only needed for this check. The curve is drawn by two lines in this case.

There is no approximation done by the curve algorithm. The error value of the last pixel will always be exactly zero. That's why the break condition of the loop is secure to test just for the last pixel. But during testing it is helpful to add an additional loop counter in the `for` statement if something goes wrong.

Since each step also modifies the increment value it had to be saved for the second test. Otherwise a few pixel would shift one y-step. This would normally not be noticed *except* if this was the last pixel for the test of the break condition.

3.4 High resolution raster

The previous algorithm approximates the Bézier curve with two lines if another part of the curve is too close to the set pixel. The plotted Bézier curve does not look smooth in such a case.

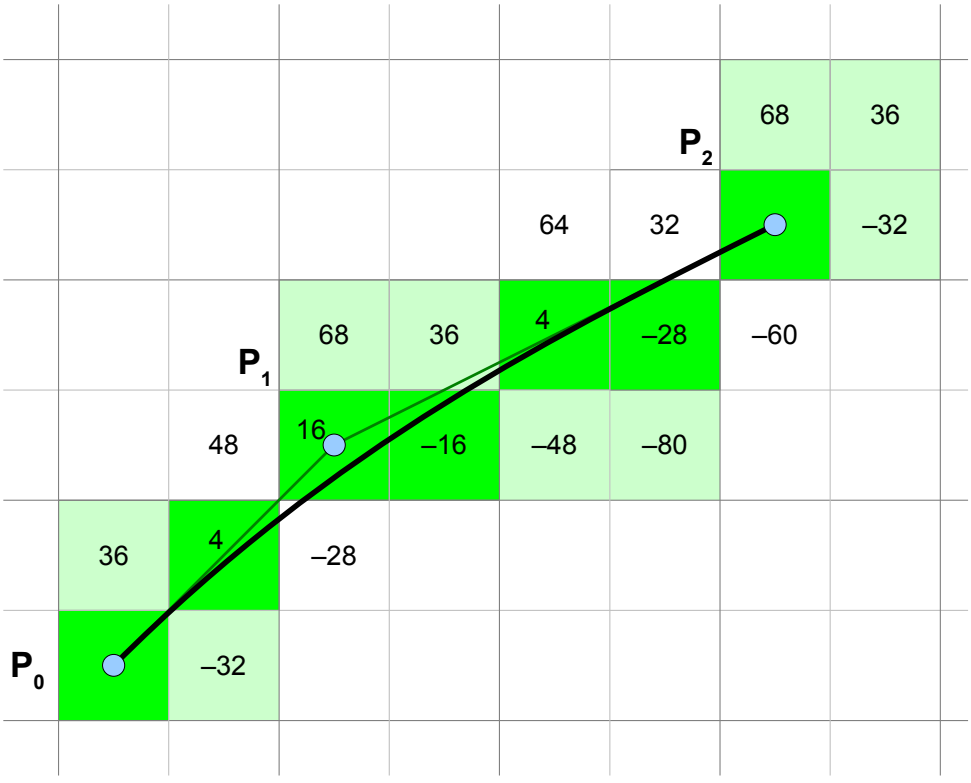


Figure 10: Higher resolution by sub-pixel raster

Another alternative is to use a finer raster of sub-pixel and set the pixel closest to this pixel curve. This high resolution raster must be sufficiently fine to avoid a conflict of two curves on one pixel or very close pixel.

Figure 10 shows a Bézier curve with a sub-pixel raster of double precision. Each pixel (light-green) is divided in sub-pixel (green).

The algorithm itself works on the finer pixel raster and has therefore no problem finding a path of suitable error values. Every time a sub-pixel is complete the pixel itself is set.

The concept of sub-pixeling is also used by the algorithm of [Emeliyanenko, 2007] to exactly draw implicit curves. This concept offers a solution if the algorithm of this document fails or gets too complicated for an implementation.

The version in listing 8 requires more computations in the pixel loop than the basic algorithm and is therefore a bit slower but never approximates the quadratic Bézier curve.

The calculation of the resolution factor makes sure that the sign of the error value does not change due to a close curve three sub-pixel from P_0 or P_2 in x or y direction. But this solution ends in a division-by-zero in case of a maximum which has to be given special care.

Since the error calculation is one pixel ahead, the computation of the last step goes beyond the end pixel. The basic algorithm has no problem stopping in this case although the increment values may already be invalid since the curve could make a sharp turn. But the fine algorithm finishes all sub-pixels of a pixel. But in case of a turn the sub-pixels cannot be finished since the increment values already changed the sign. For this case an extra break condition must be inserted in the inner loop to avoid an infinite loop.

This resolution factor can get quite large for certain Béziers. But only a few eccentric curves slow the algorithm down.

The benefit of this algorithm is that the plotted curve has no approximation errors. All set pixels are as close as possible to the analog Bézier curve.

An option is to use the fine algorithm only if the basic version fails (f >1). A long curve could also be subdivided into a long nearly straight part plotted by the faster basic version and a shorter curved part by the fine algorithm.

```
void plotFineQuadBezier(int x0, int y0, int x1, int y1, int x2, int y2)
{
  int sx = x0<x2 ? 1 : -1, sy = y0<y2 ? 1 : -1;    /* step direction */
  long f = 1, fx = x0-2*x1+x2, fy = y0-2*y1+y2;
  long long x = 2*fx*fx, y = 2*fy*fy, xy = 2*fx*fy*sx*sy;
  long long cur = sx*sy*(fx*(y2-y0)-fy*(x2-x0));        /* curvature */
                                    /* compute error increments of P0 */
  long long dx = abs(y0-y1)*xy-abs(x0-x1)*y-cur*abs(y0-y2);
  long long dy = abs(x0-x1)*xy-abs(y0-y1)*x+cur*abs(x0-x2);
                                    /* compute error increments of P2 */
  long long ex = abs(y2-y1)*xy-abs(x2-x1)*y+cur*abs(y0-y2);
  long long ey = abs(x2-x1)*xy-abs(y2-y1)*x-cur*abs(x0-x2);

                                  /* sign of gradient must not change */
  assert((x0-x1)*(x2-x1) <= 0 && (y0-y1)*(y2-y1) <= 0);

  if (cur == 0) { plotLine(x0,y0, x2,y2); return; } /* straight line */

                         /* compute required minimum resolution factor */
  if (dx == 0 || dy == 0 || ex == 0 || ey == 0)
    f = abs(xy/cur)/2+1;          /* division by zero: use curvature */
  else {
    fx = 2*y/dx; if (fx > f) f = fx;          /* increase resolution */
    fx = 2*x/dy; if (fx > f) f = fx;
    fx = 2*y/ex; if (fx > f) f = fx;
    fx = 2*x/ey; if (fx > f) f = fx;
  }                                            /* negated curvature? */
  if (cur < 0) { x = -x; y = -y; dx = -dx; dy = -dy; xy = -xy; }
  dx = f*dx+y/2-xy; dy = f*dy+x/2-xy; ex = dx+dy+xy; /* error 1.step */

  for (fx = fy = f; ; ) {                              /* plot curve */
    setPixel(x0,y0);
    if (x0 == x2 && y0 == y2) break;
    do {                                          /* move f sub-pixel */
      ey = 2*ex-dy;                  /* save value for test of y step */
      if (2*ex >= dx) { fx--; dy -= xy; ex += dx += y; }   /* x step */
      if (ey <= 0) { fy--; dx -= xy; ex += dy += x; }      /* y step */
    } while (fx > 0 && fy > 0);                    /* pixel complete? */
    if (2*fx <= f) { x0 += sx; fx += f; }  /* sufficient sub-steps.. */
    if (2*fy <= f) { y0 += sy; fy += f; }         /* .. for a pixel? */
  }
}
```

Listing 8: Plotting a Bézier curve on a fine grid

3.5 Smart curve plotting

When does the basic algorithm fail and needs special attention? Considering the entire curve the Bézier is a parabola. The algorithm fails when the two symmetric parts of the curve come too close together inside one pixel. But from this point to the vertex of the Bézier the curve becomes a straight line since the other part starts from the same pixel. The problem is similar to the ellipse algorithm. When it fails only a line is left to plot.

The algorithm could be simplified under two conditions: the drawing begins with the longer part of the curve where they are still clearly apart and secondly the remaining part is a simple line when the algorithm fails. The problem does not occur when the vertex of the parabola is drawn since the curve must be subdivided in that case.

The algorithm begins at the end which is farther away from the vertex since the other part of the curve is then probably far enough away. The algorithm stops if the two symmetric parts of the parabola come too close together and the algorithm fails. This could be tested if the derivative of the gradient of the error value changes its sign. The curve is then finished by plotting a straight line to the end of the curve.

```
void plotQuadBezierSeg(int x0, int y0, int x1, int y1, int x2, int y2)
{                               /* plot a limited quadratic Bezier segment */
  int sx = x2-x1, sy = y2-y1;
  long xx = x0-x1, yy = y0-y1, xy;         /* relative values for checks */
  double dx, dy, err, cur = xx*sy-yy*sx;                    /* curvature */

  assert(xx*sx <= 0 && yy*sy <= 0);  /* sign of gradient must not change */

  if (sx*(long)sx+sy*(long)sy > xx*xx+yy*yy) { /* begin with longer part */
    x2 = x0; x0 = sx+x1; y2 = y0; y0 = sy+y1; cur = -cur;  /* swap P0 P2 */
  }
  if (cur != 0) {                                     /* no straight line */
    xx += sx; xx *= sx = x0 < x2 ? 1 : -1;            /* x step direction */
    yy += sy; yy *= sy = y0 < y2 ? 1 : -1;            /* y step direction */
    xy = 2*xx*yy; xx *= xx; yy *= yy;          /* differences 2nd degree */
    if (cur*sx*sy < 0) {                           /* negated curvature? */
      xx = -xx; yy = -yy; xy = -xy; cur = -cur;
    }
    dx = 4.0*sy*cur*(x1-x0)+xx-xy;             /* differences 1st degree */
    dy = 4.0*sx*cur*(y0-y1)+yy-xy;
    xx += xx; yy += yy; err = dx+dy+xy;                /* error 1st step */
    do {
      setPixel(x0,y0);                                    /* plot curve */
      if (x0 == x2 && y0 == y2) return;  /* last pixel -> curve finished */
      y1 = 2*err < dx;                  /* save value for test of y step */
      if (2*err > dy) { x0 += sx; dx -= xy; err += dy += yy; } /* x step */
```

```
      if (    y1    ) { y0 += sy; dy -= xy; err += dx += xx; } /* y step */
    } while (dy < dx);            /* gradient negates -> algorithm fails */
  }
  plotLine(x0,y0, x2,y2);                  /* plot remaining part to end */
}
```

Listing 9: Fast Bézier curve algorithm

The problem does not occur when the Bézier curve consists of both parts of the parabola. If the curvature of the vertex shrinks to a single point then the curve must be subdivided before since the gradient also changes the direction there. If it is not a single point then the two sides are far enough apart and the algorithm does not fail.

This solution is very efficient to plot a quadratic Bézier curve. The algorithm has the advantage that if the curve comes too close together to work it is straight enough to be finished by a line so it never fails.

Since the error values can get quite large (up to the fourth power) the double type is used instead of a long integer. If a 64-bit integer type is available this type could be used instead.

3.6 Common Bézier curves

The previous Bézier algorithms rely on a continuously positive or negative gradient to keep it simple. A sign change would imply a change in the direction of the set pixel inside the loop. The error calculation is one pixel ahead and would need a change too. The Bézier curve is subdivided at the horizontal and vertical turns to avoid these troubles.

Subdividing the Bézier curve also has the additional benefit of avoiding unfavorable turns.

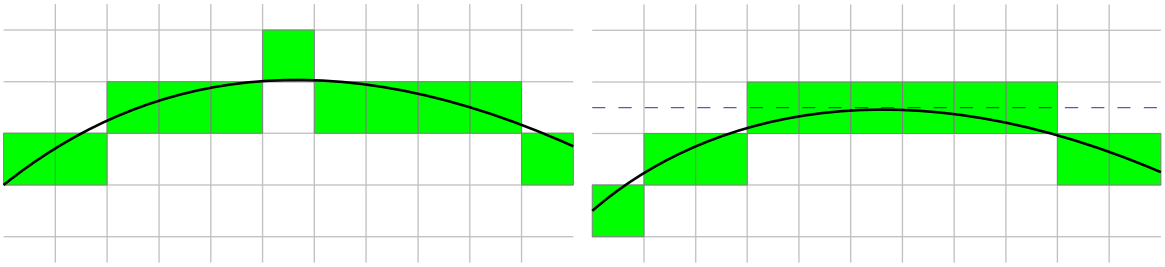


Figure 11: unfavorable curve turn

The left of figure 11 shows a curve turn with an isolated pixel which just seems at the wrong place. From the point of view of the algorithm it is right since it is closest to the curve. To avoid an unfavorable situation the turn of the curve is snapped to the center, a pixel shown in the right of figure 11 as dashed line. Of course this changes the entire curve slightly but looks much better.

The algorithm must sub-divide the Bézier curve at points at which the gradient changes its sign. These are vertical and horizontal turning points of the curve.

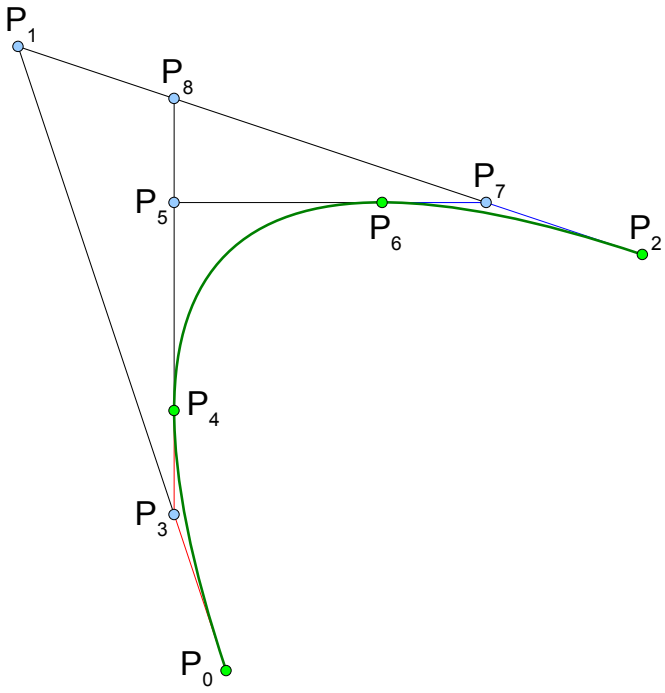


Figure 12: Subdividing a Bézier curve

Even if it is hard to believe, *one* quadratic Bézier curve could have *two* gradient changes and must be subdivided in up to *three* parts to make it plotable by the previous program. Such a more complex curve is visible in figure 12. The given points P_0, P_1 and P_2 of the curve had to be split up at the vertical point P_4 and at the horizontal point P_6 into the shorter Bézier curve $P_0/P_3/P_4$, curve $P_4/P_5/P_6$ and curve $P_6/P_7/P_2$.

Setting the gradient equation of the implicit Bézier equation to zero gets the point of the gradient change: $t(P_0 - 2P_1 + P_2) - P_0 + P_1 = 0$. Only values of $0 \le t \le 1$ indicate a change of the gradient sign. The Point P_5 is computed by $t_x = \frac{x_0 - x_1}{x_0 - 2x_1 + x_2}$, $t_y = \frac{y_0 - y_1}{y_0 - 2y_1 + y_2}$ and $x_5 = \frac{x_0 x_2 - x_1^2}{x_0 - 2x_1 + x_2}$, $y_5 = \frac{y_0 y_2 - y_1^2}{y_0 - 2y_1 + y_y}$. P_4 and P_6 are computed the same way:

$$x_6 = (1-t_y)^2 x_0 + 2(1-t_y) t_y x_1 + t_y x_2, \qquad y_4 = (1-t_x)^2 y_0 + 2(1-t_x) t_x y_1 + t_x y_2.$$

P_3 and P_7 could be computed by line intersection: $y_3 = y_0 - \frac{x_0 - x_5}{x_0 - x_1}(y_0 - y_1)$,

$$x_7 = x_2 - \frac{y_2 - y_5}{y_2 - y_1}(x_2 - x_1).$$

3.7 Program to plot any Bézier curve

The program subdivides the curve at horizontal and vertical gradient changes and delegates the plotting to the sub-procedure.

It first checks if a horizontal cut ($dx = 0$) at P_4 is necessary and if a vertical cut ($dy = 0$) occurs too. If both cuts occur it makes sure the horizontal split comes first by conditionally swapping the points. Then it plots the part and cuts it off. The same is done for the vertical cut.

It is interesting to note that the integer division of negative numbers (and reminders) depend on the application, platform or programing language. The mathematical definition (in Ruby or Python) rounds towards the lower integer value (in which the remainder always remains positive): (–5)/3 = –2 whereas the technical definition (in C/C++ or Java) rounds towards zero: (–5)/3 = –1. Please be aware that your target system may handle it differently than the example program.

```
void plotQuadBezier(int x0, int y0, int x1, int y1, int x2, int y2)
{                                   /* plot any quadratic Bezier curve */
   int x = x0-x1, y = y0-y1;
   double t = x0-2*x1+x2, r;

   if ((long)x*(x2-x1) > 0) {                /* horizontal cut at P4? */
      if ((long)y*(y2-y1) > 0)              /* vertical cut at P6 too? */
         if (fabs((y0-2*y1+y2)/t*x) > abs(y)) {       /* which first? */
            x0 = x2; x2 = x+x1; y0 = y2; y2 = y+y1;    /* swap points */
         }                      /* now horizontal cut at P4 comes first */
      t = (x0-x1)/t;
      r = (1-t)*((1-t)*y0+2.0*t*y1)+t*t*y2;               /* By(t=P4) */
      t = (x0*x2-x1*x1)*t/(x0-x1);               /* gradient dP4/dx=0 */
      x = floor(t+0.5); y = floor(r+0.5);
      r = (y1-y0)*(t-x0)/(x1-x0)+y0;          /* intersect P3 | P0 P1 */
      plotQuadBezierSeg(x0,y0, x,floor(r+0.5), x,y);
      r = (y1-y2)*(t-x2)/(x1-x2)+y2;          /* intersect P4 | P1 P2 */
      x0 = x1 = x; y0 = y; y1 = floor(r+0.5);     /* P0 = P4, P1 = P8 */
   }
   if ((long)(y0-y1)*(y2-y1) > 0) {            /* vertical cut at P6? */
      t = y0-2*y1+y2; t = (y0-y1)/t;
      r = (1-t)*((1-t)*x0+2.0*t*x1)+t*t*x2;               /* Bx(t=P6) */
      t = (y0*y2-y1*y1)*t/(y0-y1);               /* gradient dP6/dy=0 */
      x = floor(r+0.5); y = floor(t+0.5);
      r = (x1-x0)*(t-y0)/(y1-y0)+x0;          /* intersect P6 | P0 P1 */
      plotQuadBezierSeg(x0,y0, floor(r+0.5),y, x,y);
      r = (x1-x2)*(t-y2)/(y1-y2)+x2;          /* intersect P7 | P1 P2 */
      x0 = x; x1 = floor(r+0.5); y0 = y1 = y;     /* P0 = P6, P1 = P7 */
```

```
    }
    plotQuadBezierSeg(x0,y0, x1,y1, x2,y2);          /* remaining part */
}
```

Listing 10: Subdividing complex quadratic Bézier curve

Rounding in C is a bit tricky especially if it should also work for negative numbers. Floating point arithmetic is therefore only used for proper rounding of the divisions. It is also possible to do it in integers only but it is more complicated:
round(*a*/*b*) = (*a*+sign(*a*)*abs(*b*)/2)/*b*.

Sub dividing a Bézier curve always leads to certain integer rounding errors of the Bézier points. This becomes especially visible if two of the points of a Bézier come close together (about < 10 pixel). Sometimes one of the curves even becomes a straight line and does not seem to fit to the others. On the other hand this also has the benefit that horizontal and/or vertical intervals of the curve are always rasterized on full pixel. Curved transitions look much better that way.

The middle control point P_1 could be changed to a thru point $\mathring{P}_1$ by $P_1 = 2\mathring{P}_1 - \frac{P_0+P_2}{2}$. (10)

4 Rational Béziers

For rational Béziers each point P_i of equation (4) gets an additional weight w_i: [Marsh, 2005, p. 175]

$$B_n(t)=\frac{\sum_i \binom{n}{i}(1-t)^{n-i}t^i w_i P_i}{\sum_i \binom{n}{i}(1-t)^{n-i}t^i w_i}. \quad [\,0\le t\le 1\,] \tag{11}$$

4.1 Quadratic rational Béziers

For n = 2 of equation (11) the rational quadratic Bézier becomes

$$B_2(t)=\frac{(1-t)^2 w_0 P_0+2(1-t)t\,w_1 P_1+t^2 w_2 P_2}{(1-t)^2 w_0+2(1-t)t\,w_1+t^2 w_2} \tag{12}$$

The same matrix equation (7) for non-rational Béziers is used to compute the implicit quadratic rational Bézier equation except that the equation (6) for t = ½ must consider the weights:

$$\begin{aligned} a(w_0x_0+2w_1x_1+w_2x_2)^2+2b(w_0x_0+2w_1x_1+w_2x_2)(w_0y_0+2w_1y_1+w_2y_2)+\\ +c(w_0y_0+2w_1y_1+w_2y_2)^2+2d(w_0+2w_1+w_2)(w_0x_0+2w_1x_1+w_2x_2)+\\ +2e(w_0+2w_1+w_2)(w_0y_0+2w_1y_1+w_2y_2)=-(w_0+2w_1+w_2)^2 \end{aligned}$$

Solving the matrix equation the implicit equation of the rational quadratic Bézier curve makes then:

$$\begin{aligned} x^2\left(w_0w_2(\hat y_0-\hat y_2)^2+4w_1^2\hat y_0\hat y_2\right)-2xy\left(w_0w_2(\hat x_0-\hat x_2)(\hat y_0-\hat y_2)+2w_1^2(\hat x_0\hat y_2+\hat x_2\hat y_0)\right)+\\ +y^2\left(w_0w_2(\hat x_0-\hat x_2)^2+4w_1^2\hat x_0\hat x_2\right)+2w_0w_2\left(x(\hat y_0-\hat y_2)-y(\hat x_0-\hat x_2)\right)(\hat x_0\hat y_2-\hat x_2\hat y_0)+\\ +w_0w_2(\hat x_0\hat y_2-\hat x_2\hat y_0)^2=0. \end{aligned}$$

This equation suggests the substitution of $w^2=\frac{w_1^2}{w_0w_2}$ for simplification, expressing all weights only by one value.

The implicit equation of the rational quadratic Bézier becomes then: (13)

$$\begin{aligned} x^2\left((\hat y_0-\hat y_2)^2+4w^2\hat y_0\hat y_2\right)-2xy\left((\hat x_0-\hat x_2)(\hat y_0-\hat y_2)+2w^2(\hat x_0\hat y_2+\hat x_2\hat y_0)\right)+\\ +y^2\left((\hat x_0-\hat x_2)^2+4w^2\hat x_0\hat x_2\right)+2\left(x(\hat y_0-\hat y_2)-y(\hat x_0-\hat x_2)\right)(\hat x_0\hat y_2-\hat x_2\hat y_0)+(\hat x_0\hat y_2-\hat x_2\hat y_0)^2=0. \end{aligned}$$

The individual weights at the end points P_0 and P_2 of the rational quadratic Bézier change the curve the same way as the weight of the middle point P_1. The weights could therefore be normalized without changing the curve by substitution of the weight of the middle point P_1 by $w=w_1/\sqrt{w_0w_2}$. The weights of the two end points become then one and could be ignored.

For w =1 the curve is a parabola, for w < 1 the curve is an ellipse, for w = 0 the curve is a straight line and for w >1 the curve is a hyperbola. The weights are normally assumed to be all positive.

The quadratic Bézier curve must again be subdivided at horizontal and vertical turning points. These four points are calculated by setting the first derivative of the parameter equation (12) to zero:

$$t=\frac{2w(\mathrm{P}_0-\mathrm{P}_1)-\mathrm{P}_0+\mathrm{P}_2\pm\sqrt{4w^2(\mathrm{P}_0-\mathrm{P}_1)(\mathrm{P}_2-\mathrm{P}_1)+(\mathrm{P}_0-\mathrm{P}_2)^2}}{2(w-1)(\mathrm{P}_0-\mathrm{P}_2)}. \qquad (14)$$

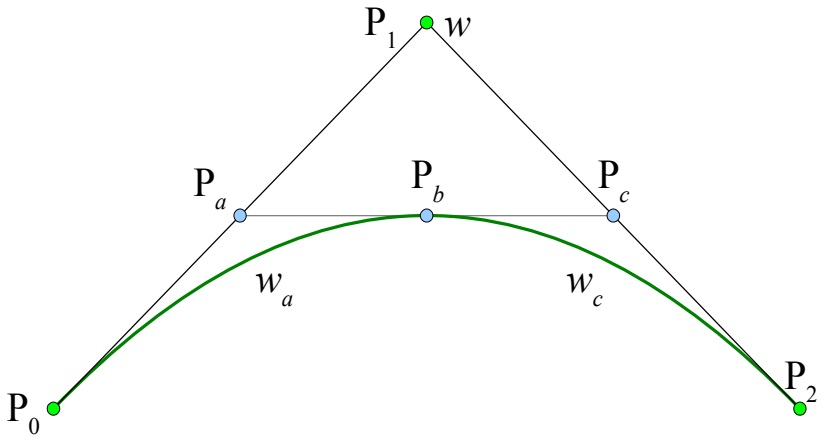


Figure 13: Subdivision of quadratic rational Bézier

The *de Casteljau* algorithm could be extended for rational Béziers by converting the 2D vector $[x_i, y_i]$ plus weight w_i of the rational curve to a 3D vector of the non-rational curve $[w_i x_i, w_i y_i, w_i]$. After the subdivision the 3D vector $[x_i, y_i, w_i]$ is mapped back to 2D $[x_i/w_i, y_i/w_i]$ plus weight w_i. When the curve in figure 13 is subdivided at the position of the parameter t then the additional points became:

$$\mathrm{P}_a=\frac{t(w\,\mathrm{P}_1-\mathrm{P}_0)+\mathrm{P}_0}{t(w-1)+1},\quad w_a=\frac{t(w-1)+1}{\sqrt{2t(1-t)(w-1)+1}},$$
$$\mathrm{P}_b=\frac{t^2(\mathrm{P}_0-2w\,\mathrm{P}_1+\mathrm{P}_2)+2t(w\,\mathrm{P}_1-\mathrm{P}_0)+\mathrm{P}_0}{2t(1-t)(w-1)+1},$$
$$\mathrm{P}_c=\frac{(1-t)(w\,\mathrm{P}_1-\mathrm{P}_2)+\mathrm{P}_2}{(1-t)(w-1)+1},\quad w_c=\frac{(1-t)(w-1)+1}{\sqrt{2t(1-t)(w-1)+1}} \qquad [0\le t\le 1] \qquad (15)$$

This time the subdivision algorithm is presented first and the drawing algorithm later. The subdivision algorithm is the same than listing 10 except that it considers the weight for the calculations. The weight mostly appears squared in rational equations. To make calculations easier the weight parameter in the segment drawing algorithm is defines as squared.

```
void plotQuadRationalBezier(int x0, int y0, int x1, int y1,
                            int x2, int y2, float w)
{                          /* plot any quadratic rational Bezier curve */
   int x = x0-2*x1+x2, y = y0-2*y1+y2;
   double xx = x0-x1, yy = y0-y1, ww, t, q;
   assert(w >= 0.0);

   if (xx*(x2-x1) > 0) {                    /* horizontal cut at P4? */
      if (yy*(y2-y1) > 0)                  /* vertical cut at P6 too? */
         if (fabs(xx*y) > fabs(yy*x)) {               /* which first? */
            x0 = x2; x2 = xx+x1; y0 = y2; y2 = yy+y1;  /* swap points */
         }                     /* now horizontal cut at P4 comes first */
      if (x0 == x2 || w == 1.0) t = (x0-x1)/(double)x;
      else {                           /* non-rational or rational case */
         q = sqrt(4.0*w*w*(x0-x1)*(x2-x1)+(x2-x0)*(long)(x2-x0));
         if (x1 < x0) q = -q;
         t = (2.0*w*(x0-x1)-x0+x2+q)/(2.0*(1.0-w)*(x2-x0));/* t at P4 */
      }
      q = 1.0/(2.0*t*(1.0-t)*(w-1.0)+1.0);         /* sub-divide at t */
      xx = (t*t*(x0-2.0*w*x1+x2)+2.0*t*(w*x1-x0)+x0)*q;       /* = P4 */
      yy = (t*t*(y0-2.0*w*y1+y2)+2.0*t*(w*y1-y0)+y0)*q;
      ww = t*(w-1.0)+1.0; ww *= ww*q;            /* squared weight P3 */
      w = ((1.0-t)*(w-1.0)+1.0)*sqrt(q);                 /* weight P8 */
      x = floor(xx+0.5); y = floor(yy+0.5);                    /* P4 */
      yy = (xx-x0)*(y1-y0)/(x1-x0)+y0;        /* intersect P3 | P0 P1 */
      plotQuadRationalBezierSeg(x0,y0, x,floor(yy+0.5), x,y, ww);
      yy = (xx-x2)*(y1-y2)/(x1-x2)+y2;        /* intersect P4 | P1 P2 */
      y1 = floor(yy+0.5); x0 = x1 = x; y0 = y;    /* P0 = P4, P1 = P8 */
   }
   if ((y0-y1)*(long)(y2-y1) > 0) {            /* vertical cut at P6? */
      if (y0 == y2 || w == 1.0) t = (y0-y1)/(y0-2.0*y1+y2);
      else {                           /* non-rational or rational case */
         q = sqrt(4.0*w*w*(y0-y1)*(y2-y1)+(y2-y0)*(long)(y2-y0));
         if (y1 < y0) q = -q;
         t = (2.0*w*(y0-y1)-y0+y2+q)/(2.0*(1.0-w)*(y2-y0));/* t at P6 */
      }
      q = 1.0/(2.0*t*(1.0-t)*(w-1.0)+1.0);         /* sub-divide at t */
      xx = (t*t*(x0-2.0*w*x1+x2)+2.0*t*(w*x1-x0)+x0)*q;       /* = P6 */
      yy = (t*t*(y0-2.0*w*y1+y2)+2.0*t*(w*y1-y0)+y0)*q;
      ww = t*(w-1.0)+1.0; ww *= ww*q;            /* squared weight P5 */
      w = ((1.0-t)*(w-1.0)+1.0)*sqrt(q);                 /* weight P7 */
      x = floor(xx+0.5); y = floor(yy+0.5);                    /* P6 */
      xx = (x1-x0)*(yy-y0)/(y1-y0)+x0;        /* intersect P6 | P0 P1 */
      plotQuadRationalBezierSeg(x0,y0, floor(xx+0.5),y, x,y, ww);
      xx = (x1-x2)*(yy-y2)/(y1-y2)+x2;        /* intersect P7 | P1 P2 */
      x1 = floor(xx+0.5); x0 = x; y0 = y1 = y;    /* P0 = P6, P1 = P7 */
   }
   plotQuadRationalBezierSeg(x0,y0, x1,y1, x2,y2, w*w);  /* remaining */
}
```

Listing 11: Subdividing a quadratic rational Bézier curve

The program of listing 11 subdivides a quadratic rational Bézier curve at the horizontal and vertical turning points the same way as listing 10 does for non-rational Béziers. The remarks in listing 11 refer therefore to figure 12.

This implementation also plots non-rational Béziers.

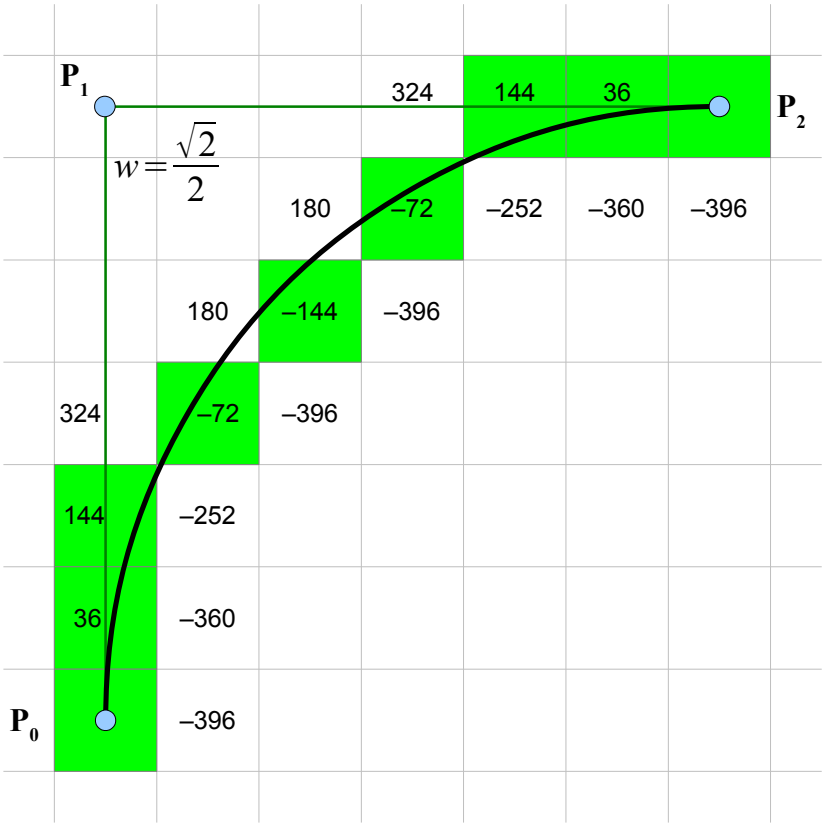


Figure 14: Error values of a quadratic rational Bézier

The values of the error calculation make:

$$d_x = \mathrm{e}(\hat{x}_0+1,\hat{y}_0) = 4w^2\hat{y}_0(\hat{x}_2\hat{y}_0-\hat{x}_0\hat{y}_2-\hat{y}_2)-(\hat{y}_2-\hat{y}_0)^2, \tag{16}$$

$$d_y = \mathrm{e}(\hat{x}_0,\hat{y}_0+1) = 4w^2\hat{x}_0(\hat{x}_0\hat{y}_2-\hat{x}_2\hat{y}_0-\hat{x}_2)-(\hat{x}_2-\hat{x}_0)^2,$$

$$d_{xx} = \frac{\partial^2 \mathrm{e}}{\partial x^2} = -2(4w^2\hat{y}_0\hat{y}_2+(\hat{y}_2-\hat{y}_0)^2),\quad d_{yy} = \frac{\partial^2 \mathrm{e}}{\partial y^2} = -2(4w^2\hat{x}_0\hat{x}_2+(\hat{x}_2-\hat{x}_0)^2),$$

$$d_{xy} = \frac{\partial^2 \mathrm{e}}{\partial x\,\partial y} = 2(2w^2(\hat{x}_0\hat{y}_2+\hat{x}_2\hat{y}_0)+(\hat{x}_2-\hat{x}_0)(\hat{y}_2-\hat{y}_0)).$$

For $w = 1$ these values equal the equations of chapter 3.1.

4.2 Rational quadratic algorithm

Very small values of the weight may cause the algorithm to fail. The reason is the same than before. The other part of the ellipse comes too close. Although this is a rare case an easy fix is simply to subdivide the curve in half and plot each segment separately.

```
void plotQuadRationalBezierSeg(int x0, int y0, int x1, int y1,
                               int x2, int y2, float w)
{                  /* plot a limited rational Bezier segment, squared weight */
  int sx = x2-x1, sy = y2-y1;                    /* relative values for checks */
  double dx = x0-x2, dy = y0-y2, xx = x0-x1, yy = y0-y1;
  double xy = xx*sy+yy*sx, cur = xx*sy-yy*sx, err;               /* curvature */

  assert(xx*sx <= 0.0 && yy*sy <= 0.0);   /* sign of gradient must not change */

  if (cur != 0.0 && w > 0.0) {                              /* no straight line */
    if (sx*(long)sx+sy*(long)sy > xx*xx+yy*yy) {    /* begin with longer part */
      x2 = x0; x0 -= dx; y2 = y0; y0 -= dy; cur = -cur;         /* swap P0 P2 */
    }
    xx = 2.0*(4.0*w*sx*xx+dx*dx);                   /* differences 2nd degree */
    yy = 2.0*(4.0*w*sy*yy+dy*dy);
    sx = x0 < x2 ? 1 : -1;                                  /* x step direction */
    sy = y0 < y2 ? 1 : -1;                                  /* y step direction */
    xy = -2.0*sx*sy*(2.0*w*xy+dx*dy);

    if (cur*sx*sy < 0.0) {                                /* negated curvature? */
      xx = -xx; yy = -yy; xy = -xy; cur = -cur;
    }
    dx = 4.0*w*(x1-x0)*sy*cur+xx/2.0+xy;            /* differences 1st degree */
    dy = 4.0*w*(y0-y1)*sx*cur+yy/2.0+xy;

    if (w < 0.5 && dy > dx) {                 /* flat ellipse, algorithm fails */
       cur = (w+1.0)/2.0; w = sqrt(w); xy = 1.0/(w+1.0);
       sx = floor((x0+2.0*w*x1+x2)*xy/2.0+0.5);    /* subdivide curve in half */
       sy = floor((y0+2.0*w*y1+y2)*xy/2.0+0.5);
       dx = floor((w*x1+x0)*xy+0.5); dy = floor((y1*w+y0)*xy+0.5);
       plotQuadRationalBezierSeg(x0,y0, dx,dy, sx,sy, cur);/* plot separately */
       dx = floor((w*x1+x2)*xy+0.5); dy = floor((y1*w+y2)*xy+0.5);
       plotQuadRationalBezierSeg(sx,sy, dx,dy, x2,y2, cur);
       return;
    }
    err = dx+dy-xy;                                          /* error 1.step */
    do {
      setPixel(x0,y0);                                        /* plot curve */
      if (x0 == x2 && y0 == y2) return;       /* last pixel -> curve finished */
      x1 = 2*err > dy; y1 = 2*(err+yy) < -dy;/* save value for test of x step */
      if (2*err < dx || y1) { y0 += sy; dy += xy; err += dx += xx; }/* y step */
      if (2*err > dx || x1) { x0 += sx; dx += xy; err += dy += yy; }/* x step */
    } while (dy < dx);                 /* gradient negates -> algorithm fails */
  }
  plotLine(x0,y0, x2,y2);                     /* plot remaining needle to end */
}
```

Listing 12: Plot a limited rational Bezier segment

This algorithm also avoids setting spurious pixel. The pixel loop of listing 12 therefore looks if no second x or y step happens and takes the appropriate step beforehand. Such a check could be included in every pixel loop.

4.3 Rotating the ellipse

Now the tools for solving the problem of the rotated ellipse are developed.

When the ellipse is transformed by the rotation matrix $R(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$ the implicit equation becomes:

$$x^2(a^2\sin^2\theta+b^2\cos^2\theta)-2xy(a^2-b^2)\sin\theta\cos\theta+y^2(a^2\cos^2\theta+b^2\sin^2\theta)-a^2b^2=0.$$

With the definitions of $x_d^2=a^2\cos^2\theta+b^2\sin^2\theta$, $y_d^2=a^2\sin^2\theta+b^2\cos^2\theta$ and $z_d=(a^2-b^2)\sin\theta\cos\theta=(a^2-b^2)\frac{\tan\theta}{1+\tan^2\theta}=\frac{a^2e^2}{2}\sin 2\theta$ (e .. eccentricity) the implicit equation of the ellipse rotated by the angle θ becomes:

$$x^2y_d^2-2xyz_d+y^2x_d^2-x_d^2y_d^2+z_d^2=0 \qquad [|z_d|\le x_d y_d] \tag{17}$$

x_d and y_d are the size of new rectangle enclosing the rotated ellipse. If $|z_d|$ equals $x_d y_d$ then the ellipse becomes a straight diagonal line.

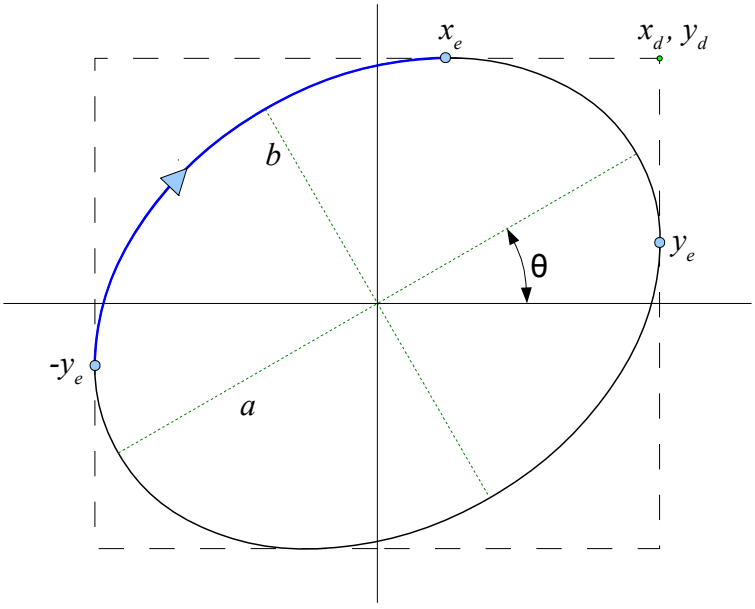


Figure 15: Rotated ellipse

The values of the maximums (contact points to the rectangle) make $x_e y_d = x_d y_e = z_d$.

Useful relations: $2a^2=x_d^2+y_d^2+\sqrt{(x_d^2-y_d^2)^2+4z_d^2}$, $2b^2=x_d^2+y_d^2-\sqrt{(x_d^2-y_d^2)^2+4z_d^2}$.

The algorithm is divided into two parts. The first part calculates x_d, y_d, z_d and calls the second part by the parameters of corners of the rectangle. This proceeding snaps the curve of the ellipse to integer values and makes it again possible of drawing rotated ellipses of odd diameters or without calculating trigonometric functions.

4.4 Rational Bézier ellipses

The algorithm faces the same problem as quadratic Bézier curves. If the second half of the flat ellipse comes too close the algorithm fails. The smart solution of the previous implementation could not be used this time since the ellipse is a closed curve. The problems arise on the two ends of the flat ellipse. To use the same solution the algorithm would have to start at the middle and plot towards its narrow ends. This would become a complex program. Another approach developed earlier is to increase the resolution of the raster by a finer grid. This high resolution raster must be sufficiently fine to avoid a conflict of two curves on one pixel or very close pixel.

Another solution to this problem is to use the already existing rational Bézier algorithm. An ellipse can also be regarded as a composition of four rational Bézier curves. By comparing the terms of the two implicit equations of ellipse and rational Bézier the weight of P_1 is calculated by $w^2=\frac{x_d\,y_d \pm z_d}{2\,x_d\,y_d}$ for the long/short side of the elliptic segment.

The program for drawing rotated ellipses can now delegate the drawing process to the subroutine.

```
void plotRotatedEllipse(int x, int y, int a, int b, float angle)
{                              /* plot ellipse rotated by angle (radian) */
   float xd = (long)a*a, yd = (long)b*b;
   float s = sin(angle), zd = (xd-yd)*s;           /* ellipse rotation */
   xd = sqrt(xd-zd*s), yd = sqrt(yd+zd*s);    /* surrounding rectangle */
   a = xd+0.5; b = yd+0.5; zd = zd*a*b/(xd*yd);    /* scale to integer */
   plotRotatedEllipseRect(x-a,y-b, x+a,y+b, (long)(4*zd*cos(angle)));
}

void plotRotatedEllipseRect(int x0, int y0, int x1, int y1, long zd)
{           /* rectangle enclosing the ellipse, integer rotation angle */
   int xd = x1-x0, yd = y1-y0;
   float w = xd*(long)yd;
   if (zd == 0) return plotEllipseRect(x0,y0, x1,y1);   /* looks nicer */
   if (w != 0.0) w = (w-zd)/(w+w);             /* squared weight of P1 */
   assert(w <= 1.0 && w >= 0.0);          /* limit angle to |zd|<=xd*yd */
   xd = floor(xd*w+0.5); yd = floor(yd*w+0.5);    /* snap xe,ye to int */
   plotQuadRationalBezierSeg(x0,y0+yd, x0,y0, x0+xd,y0, 1.0-w);
   plotQuadRationalBezierSeg(x0,y0+yd, x0,y1, x1-xd,y1, w);
   plotQuadRationalBezierSeg(x1,y1-yd, x1,y1, x1-xd,y1, 1.0-w);
   plotQuadRationalBezierSeg(x1,y1-yd, x1,y0, x0+xd,y0, w);
}
```

Listing 13: Programs to plot rotated ellipses

The program in listing 13 even works if P_0 and P_1 are interchanged.

The only disadvantage of this solution is that the ellipse is not always exactly symmetric. Sometimes the curve is just between two pixels and the drawing algorithm always rounds to same direction (the lower pixel for example). In case of the ellipse this means that the one side of the segment is rounded inwards and the other symmetric side outwards.

5 Cubic Bézier curves

Are you ready for the third degree? Rasterizing cubic curves needs a bit more mathematics. Cubic Béziers can get quite complex as figure 16 shows. The loops and cusps cause many troubles. It may not be a bad decision to subdivide the cubic curve in short curves by *de Casteljau's* algorithm and convert them to quadratic Bézier curves which could be plot by the previous algorithm.

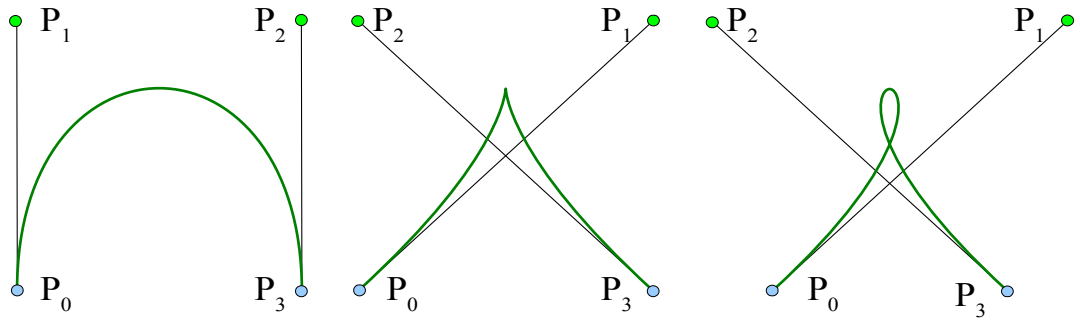


Figure 16: Various cubic Bézier curves

5.1 Cubic degree reduction

In normal cases it is not possible to exactly reduce the degree of a Bézier curve. Only if the cubic term vanishes equation (29) could be used to get a reduced quadratic equation.

It is however possible to approximate a cubic Bézier by quadratic Béziers. To keep the deviation small it is advisable to take the subdivision in three steps:

- the curve is subdivided at horizontal and vertical stationary points (chapter 5.9)
- the curve is subdivided at the inflection point (chapter 5.7)
- the remaining curve is subdivided in two quadratic Béziers

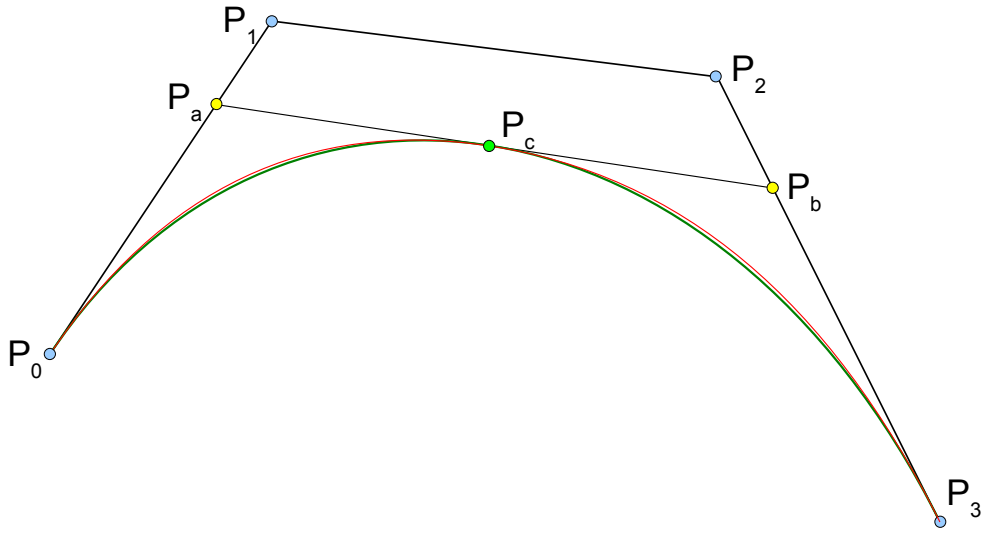


Figure 17: Approximation of a cubic Bézier (red) by two quadratic ones (green)

Of course there are a few possibilities for degree reduction and subdivision. An accurate one makes sure that the curvature at the end points of the approximation does not change which has also the benefit of easy calculations.

The following considerations keep the deviations small. The cubic Bézier P_0-P_1-P_2-P_3 in figure 17 is exactly halved (at t = ½) by DeCasteljou subdivision. The subdivision point P_c remains exactly at the cubic Bézier curve. Now the tangents at the end points P_0 and P_3 of the subdivided cubic and the two quadratic curves P_0-P_a-P_c and P_c-P_b-P_3 are made equal in direction and magnitude.

If the cubic Bézier segment P_0-P_a-P_c is obtained by the DeCasteljou subdivision the tangent at P_0 makes $P'(0) = 3(P_1-P_0)/2$ and the tangent of the quadratic Bézier P_0-P_a-P_c makes $2(P_a-P_0)$. Making this two tangent equal the points became:

$P_a = (P_0 + 3P_1)/4$, $P_b = (3P_2 + P_4)/4$ and $P_c = (P_a+P_b)/2$. (18)

If the approximation by quadratic Bézier curves is not accurate enough the cubic curve must be rasterized.

5.2 Polynomial Resultants

Again the implicit equation of the cubic Bézier curve is needed: [Marsh, 2005]

$$B_3(t)=(1-t)^3 P_0+3(1-t)^2 t P_1+3(1-t)t^2 P_2+t^3 P_3. \quad (19)$$

For deriving the general implicit equation of third order of

$$a x^3-3 b x^2 y+3 c x y^2-d y^3+3 e x^2-3 f x y+3 g y^2+3 h x-3 i y+j=0 \quad (20)$$

ten linearly independent equations are needed. This seems far too complex so lets try a different approach.

Resultants are a powerful tool to calculate the common roots of two (or more) polynomials without the laborious work of finding all individual roots. The main idea is to find a set of linearly independent polynomials to apply the theory of linear system of equations. If f_i and g_j are all the roots of the two polynomials $\mathrm{f}(t)=\sum a_i t^i=0$ and $\mathrm{g}(t)=\sum b_i t^i=0$ then the resultant is defined as the product of the differences of their roots $\mathrm{R}(f,g)=\prod(f_i-g_j)$ and could be calculated by the determinant of the Sylvester or Bézout matrix. [Bézout, 1764]

Bézout first noticed that *'… a determinate equation can always be viewed as the result of two equations in two unknowns, when one of the unknowns is eliminated'.* (Leibniz already used resultants although he never published his findings.) The problem is also related to Euclid's GCD algorithm or Gauss elimination.

If the two polynomials $\mathrm{f}(t)$ and $\mathrm{g}(t)$ have a common root t_0 then the equation $\mathrm{f}(t)\,\mathrm{g}(s)-\mathrm{f}(s)\,\mathrm{g}(t)=0$ will always be satisfied for any value of s. The equation must also contain $(s-t)$ as a factor since it will be satisfied by $t=s$ too even if there is no common root. After dividing by $(s-t)$ the equation could be seen as polynomial in s where the coefficient of each term is a polynomial in t. Since at the common root $t=t_0$ the entire expression must vanish for any value of s, each of the coefficient polynomials in t must vanish at t_0. [Cayley, 1867]

The coefficient matrix for two univariate polynomials is calculated by the Cayley expression

$$\Delta=\frac{\mathrm{f}(t)\,\mathrm{g}(s)-\mathrm{f}(s)\,\mathrm{g}(t)}{s-t}=\sum_{j=0}^{n-1}\left(\sum_{i=0}^{n-1}c_{ij}\,t^i\right)s^j=0 \quad \text{with} \tag{21}$$

$$c_{ij}=\sum_{k=\max(0,i-j)}^{\min(i,n-j-1)}\left(a_{i-k}\,b_{j+k+1}-a_{j+k+1}\,b_{i-k}\right)=c_{ji}.$$

This equation can also be written in matrix form

$$\Delta=\begin{bmatrix}1\\ \vdots\\ s^{n-1}\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}c_{00} & \cdots & c_{0,n-1}\\ \vdots & \ddots & \vdots\\ c_{n-1,0} & \cdots & c_{n-1,n-1}\end{bmatrix}\begin{bmatrix}1\\ \vdots\\ t^{n-1}\end{bmatrix}=0.$$

The square matrix of this equation is called Bézout matrix of the polynomials $\mathrm{polynomials}\ \mathrm{f}(t)\ \mathrm{and}\ \mathrm{g}(t)$. The determinant of this matrix only equals zero if a common root exists. Since the expression must vanish for any value of s the resultant is created by the coefficient matrix of n homogeneous linear equations.

A common root $t=t_0$ only exists if $\begin{bmatrix}c_{00} & \cdots & c_{0,n-1}\\ \vdots & \ddots & \vdots\\ c_{n-1,0} & \cdots & c_{n-1,n-1}\end{bmatrix}\begin{bmatrix}1\\ \vdots\\ t^{n-1}\end{bmatrix}=0$ (22)

It may be surprising to view this as linear equations but in this case the unknowns are simply powers of t and the resultant will be identically equal to zero if and only if the coefficient matrix of n homogeneous linear equations of the power of t are all zero.

The simplest example is of degree one: $\mathrm{f}(t)=a_1 t+a_0=0,\ \mathrm{g}(t)=b_1 t+b_0=0.$

$$\frac{(a_1 t+a_0)(b_1 s+b_0)-(a_1 s+a_0)(b_1 t+b_0)}{s-t}=c_{00}=a_0 b_1-a_1 b_0=0.$$

The resultant of degree two is calculated by

$$\mathrm{f}(t)=a_2 t^2+a_1 t+a_0=0,\ \mathrm{g}(t)=b_2 t^2+b_1 t+b_0=0.$$

$$\frac{(a_2 t^2+a_1 t+a_0)(b_2 s^2+b_1 s+b_0)-(a_2 s^2+a_1 s+a_0)(b_2 t^2+b_1 t+b_0)}{s-t}=(c_{11} t+c_{01})s+(c_{01} t+c_{00})=0.$$

Bézout's matrix makes then $\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix}\begin{bmatrix} 1 \\ t \end{bmatrix}=\begin{bmatrix} a_0 b_1-a_1 b_0 & a_0 b_2-a_2 b_0 \\ a_0 b_2-a_2 b_0 & a_1 b_2-a_2 b_1 \end{bmatrix}\begin{bmatrix} 1 \\ t \end{bmatrix}=0.$

The resultant of two cubic polynomials is calculated by

$$\mathrm{f}(t)=a_3 t^3+a_2 t^2+a_1 t+a_0=0,\ \mathrm{g}(t)=b_3 t^3+b_2 t^2+b_1 t+b_0=0.$$

$$\frac{\mathrm{f}(t)\mathrm{g}(s)-\mathrm{f}(s)\mathrm{g}(t)}{s-t}=(c_{22} t^2+c_{12} t+c_{02})s^2+(c_{12} t^2+c_{11} t+c_{10})s+(c_{02} t^2+c_{01} t+c_{00})=0.$$

The resultant of Bézout's matrix is

$$\begin{bmatrix} a_0 b_1-a_1 b_0 & a_0 b_2-a_2 b_0 & a_0 b_3-a_3 b_0 \\ a_0 b_2-a_2 b_0 & a_1 b_2-a_2 b_1+a_0 b_3-a_3 b_0 & a_1 b_3-a_3 b_1 \\ a_0 b_3-a_3 b_0 & a_1 b_3-a_3 b_1 & a_2 b_3-a_3 b_2 \end{bmatrix}\begin{bmatrix} 1 \\ t \\ t^2 \end{bmatrix}=0.$$

5.3 Implicit cubic Bézier equation

Now resultants can also be applied to convert the parametric equations of a curve to the implicit form $\mathrm{f}(x,y)=0$ for any value of t: $\mathrm{R}(\mathrm{f},\mathrm{g})=\begin{vmatrix} c_{00} & \cdots & c_{0,n-1} \\ \vdots & \ddots & \vdots \\ c_{n-1,0} & \cdots & c_{n-1,n-1} \end{vmatrix}=0.$ (23)

The equation of the rational Bézier curve of degree n makes

$$\mathrm{B}_n(t)=\frac{\sum_i w_i\binom{n}{i}(1-t)^{n-i}t^i \mathrm{P}_i}{\sum_i w_i\binom{n}{i}(1-t)^{n-i}t^i}. \quad (24)$$

The common root of the two parametric polynomials

$$\mathrm{f}_x(t)=\sum_i w_i\binom{n}{i}(1-t)^{(n-i)}t^i(x-x_i)=0 \quad \text{and} \quad \mathrm{f}_y(t)=\sum_i w_i\binom{n}{i}(1-t)^{(n-i)}t^i(y-y_i)=0$$

build the implicit equation of the Bézier curve $\mathrm{f}(x,y)=0$ in x and y.

$f_x(t)$ is a polynomial in t whose coefficients are linear in x, and $f_y(t)$ a polynomial in t whose coefficients are linear in y. Any value of x and y for which $f(x,y)=0$ makes the resultant of $f_x(t)$ and $f_y(t)$ to equal zero and are therefore part of the parametric curve.

The Bernstein form of the polynomial must be converted to the power form by

$$\sum_i w_i \binom{n}{i} (1-t)^{n-i} t^i = \sum_i \binom{n}{i} \left(\sum_k w_k \binom{i}{k} (-1)^{i-k} \right) t^i$$

The coefficients a_i and b_i of the polynomials $f(t)$ and $g(t)$ of the matrix (22) could be further simplified due to the linearity of determinants. Adding to each row and column m of the determinant all previous rows and columns this sum becomes

$$\sum_i \binom{m}{i} \sum_k w_k \binom{i}{k} (-1)^{i-k} (x - x_k) = w_m (x - x_m).$$

The polynomial matrix coefficients of the Bézier curve make then

$$a_i = w_i \binom{n}{i} (x - x_i) \quad \text{and} \quad b_i = w_i \binom{n}{i} (y - y_i).$$

The third degree of the cubic Bézier curve could now be implicitized as follows

$$f(x,y) = \begin{vmatrix} d_{01} & d_{02} & d_{03} \\ d_{02} & d_{03}+d_{12} & d_{13} \\ d_{03} & d_{13} & d_{23} \end{vmatrix} = 0 \quad \text{where} \quad d_{ij} = \binom{3}{i}\binom{3}{j} \begin{vmatrix} x & y & 1 \\ x_i & y_i & 1 \\ x_j & y_j & 1 \end{vmatrix} \tag{25}$$

Unfortunately the terms of this implicit equation (20) are rather messy, for example

$$a = 9\,y_{01}(y_{03}\,y_{23} + 9\,y_{12}\,y_{23} - 3\,y_{13}^2) - 27\,y_{02}^2\,y_{23} + 18\,y_{02}\,y_{03}\,y_{13} - y_{03}^2(y_{03} + 9\,y_{12})$$

where $y_{ij} = y_i - y_j$. Is there a more simple expression possible?

The key to success is symmetry. But the definition of the Bézier curves is asymmetric; the parameter t ranging from zero to one.

This has to be changed to the symmetrically defined range of the parameter from minus one half to plus one half:

$$B_3(\hat{t}) = (½ - \hat{t})^3 P_0 + 3(½ - \hat{t})^2(½ + \hat{t}) P_1 + 3(½ - \hat{t})(½ + \hat{t})^2 P_2 + (½ + \hat{t})^3 P_3 \quad \text{for} \quad [-½ \le \hat{t} \le ½].$$

To make calculations easier the constants of the cubic Bézier equation are transformed by:

$$\begin{bmatrix} x_a \\ 2\,x_b \\ 4\,x_c \\ 8\,x_d \end{bmatrix} = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 3 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} y_a \\ 2\,y_b \\ 4\,y_c \\ 8\,y_d \end{bmatrix} = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 3 & 3 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{26}$$

The Bézier equations become:

$$x = -\hat{t}^3 x_a + 3\hat{t}^2 x_b - 3\hat{t}\,x_c + x_d \quad \text{and} \quad y = -\hat{t}^3 y_a + 3\hat{t}^2 y_b - 3\hat{t}\,y_c + y_d.$$

The reverse transformation makes:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 3 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} x_a/8 \\ x_b/4 \\ x_c/2 \\ x_d \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 3 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} y_a/8 \\ y_b/4 \\ y_c/2 \\ y_d \end{bmatrix}$$

The expressions became further simpler by introducing the constants

$$c_{ij} = \begin{vmatrix} x_i & x_j \\ y_i & y_j \end{vmatrix} = x_i y_j - x_j y_i . \tag{27}$$

The terms of the implicit cubic Bézier equation (20) of $\mathrm{f}(x,y)=0$ are then computed by:

$$\begin{aligned}
&a = y_a^3, \quad b = x_a y_a^2, \quad c = x_a^2 y_a, \quad d = x_a^3, \quad e = c_{ad} y_a^2 - 3(2 c_{ab} y_c + c_{ac} y_b) y_a + 9 c_{ab} y_b^2, \\
&f = (2 c_{ad} + 3 c_{bc}) x_a y_a + 9(2 c_{ab} x_b y_b + x_b x_c y_a^2 - x_a^2 y_b y_c), \\
&g = c_{ad} x_a^2 - 3(2 c_{ab} x_c + c_{ac} x_b) x_a + 9 c_{ab} x_b^2, \\
&h = c_{ad}^2 y_a + 6(3 c_{ab} c_{bd} - c_{ac} c_{ad}) y_b + 3(3 c_{ac}^2 - 4 c_{ab} c_{ad} - 9 c_{ab} c_{bc}) y_c + 9 c_{ab} c_{ac} y_d, \\
&i = c_{ad}^2 x_a + 6(3 c_{ab} c_{bd} - c_{ac} c_{ad}) x_b + 3(3 c_{ac}^2 - 4 c_{ab} c_{ad} - 9 c_{ab} c_{bc}) x_c + 9 c_{ab} c_{ac} x_d \\
&\text{and} \quad j = c_{ad}^3 - 9 c_{ad}(c_{ac} c_{bd} + 2 c_{ab} c_{cd}) + 27(c_{ab} c_{bd}^2 + c_{ac}^2 c_{cd}) - 81 c_{ab} c_{bc} c_{cd} .
\end{aligned} \tag{28}$$

The last term j is not needed for the algorithm.

Most characteristic properties of the Bézier curve can only be defined by the values of c_{ab}, c_{ac} and c_{bc}. Please note that c_{ij} changes the sign if i and j (or x and y) are interchanged.

If x_a and y_a equal zero then the cubic term of the Bézier equation vanishes and the curve becomes the reduced quadratic (or even less) form:

$$\mathrm{B}_3(t) = (1-t)^2 \mathrm{P}_3 + (1-t) t (3 \mathrm{P}_1 - \mathrm{P}_0) + t^2 \mathrm{P}_0 \tag{29}$$

(exact degree reduction).

5.4 Cubic error calculation

The equation of the error calculation for the cubic Bézier curve is

$$\mathrm{e}(x, y) = a x^3 - 3 b x^2 y + 3 c x y^2 - d y^3 + 3 e x^2 - 3 f x y + 3 g y^2 + 3 h x - 3 i y + j$$

Fortunately this complex expression only needs to be evaluated for the initialization. During the pixel loop the calculations only need additions due to forward differencing.

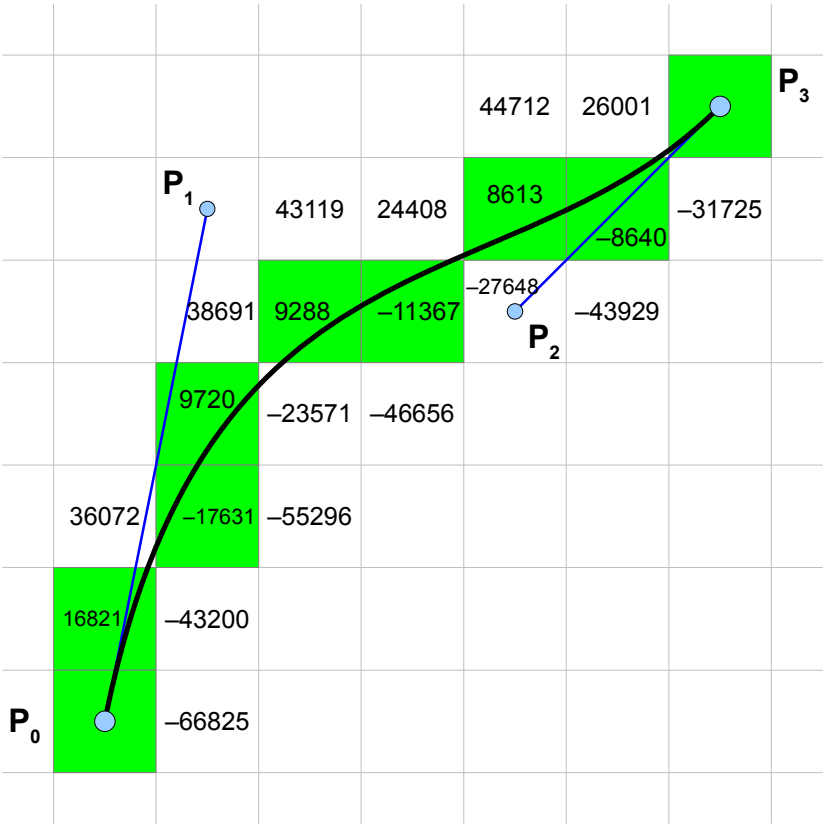


Figure 18: Error values of a cubic Bézier curve

The increment steps in x-direction make: $err \mathrel{+}= d_x$, $d_x \mathrel{+}= d_{xx}$, $d_y \mathrel{+}= d_{xy}$,

$$d_{xx} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial x^3} = 6a = 6y_a^3, \quad d_{xy} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial x^2 \partial y} = -6b = -6x_a y_a^2, \quad d_{yy} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial x \partial y^2} = 6c = 6x_a^2 y_a.$$

And in in y-direction: $err \mathrel{+}= d_y$, $d_y \mathrel{+}= d_{yy}$, $d_x \mathrel{+}= d_{xy}$,

$$d_{yy} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial y^3} = -6d = -6x_a^3, \quad d_{xy} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial x \partial y^2} = 6c = 6x_a^2 y_a, \quad d_{xx} \mathrel{+}= \frac{\partial^3 \mathrm{e}}{\partial x^2 \partial y} = -6b = -6x_a y_a^2.$$

Calculating the differences of the first and second order the forward differences follow a regular pattern. The differences of the error values are the finite *n*-th order forward differences in two dimensions:

$$d_{n_x n_y} = \sum_{i_x=0}^{n_x} \sum_{i_y=0}^{n_y} (-1)^{i_x+i_y} \binom{n_x}{i_x} \binom{n_y}{i_y} \mathrm{e}(x+n_x-i_x, y+n_y-i_y) \quad [n_x+n_y=n]. \tag{30}$$

The initialization values of the increment variables for P_0 make the finite first and second order forward differences: (31)

$$d_x = \mathrm{e}(x_0+1, y_0) = 3(a x_0(x_0+1) + y_0(c y_0 - f) - (2x_0+1)(b y_0 - e) + h) + a =$$
$$= 27(y_a + 2y_b + y_c)(c_{ab}^2 + c_{ab}c_{ac} - 3c_{ab}c_{bc} + c_{ac}^2) + 27(y_b^2 - y_a y_c)c_{ab} - 9y_a^2(c_{ab} + c_{ac} + c_{bc}) + y_a^3,$$
$$d_y = \mathrm{e}(x_0, y_0+1) = -3(d y_0(y_0+1) + x_0(b x_0 + f) - (2y_0+1)(c x_0 + g) + i) - d =$$

$$=-27(x_a+2x_b+x_c)(c_{ab}^2+c_{ab}c_{ac}-3c_{ab}c_{bc}+c_{ac}^2)+27(x_b^2-x_a x_c)c_{ab}-9x_a^2(c_{ab}+c_{ac}+c_{bc})-x_a^3,$$

$$d_{xx}=\mathrm{e}(x_0+2,y_0)-2\mathrm{e}(x_0+1,y_0)=6(a(x_0+1)-by_0+e)=$$
$$=6(y_a^3-3y_a^2(c_{ab}+c_{ac})-3y_a(2c_{ab}y_c+c_{ac}y_b)+9c_{ab}y_b^2),$$

$$d_{xy}=\mathrm{e}(x_0+1,y_0+1)-\mathrm{e}(x_0+1,y_0)-\mathrm{e}(x_0,y_0+1)=-3(b(2x_0+1)-c(2y_0+1)+f)=$$
$$=3(x_a y_a(6(c_{ab}+c_{ac})-3c_{bc}+x_a-y_a)-9(x_b x_c y_a^2-x_a^2 y_b y_c+2c_{ab}x_b y_b)),$$

$$d_{yy}=\mathrm{e}(x_0,y_0+2)-2\mathrm{e}(x_0,y_0+1)=6(cx_0-d(y_0+1)+g)=$$
$$=6(x_a^3-3x_a^2(c_{ab}+c_{ac})-3x_a(2c_{ab}x_c+c_{ac}x_b)+9c_{ab}x_b^2).$$

These values do not depend on x_d or y_d. The error calculation could be done in integers. But the values can get quite large (up to the sixth power). Variables d_{xx}, d_{xy}, d_{yy} need four times of the initial word size, d_x and d_y five times and the error value six times.

5.5 Self-intersection point

For the crunode or self-intersection point we want to find the two corresponding parameters t_1 and t_2 such that $B_3(t_1) = B_3(t_2)$, $t_1 \neq t_2$.

The problem of the self-intersection point could be solved by sub-dividing or root finding. But there is also an algebraic solution of this problem.

The self-intersection point is a saddle point where the derivative of the implicit equation of the curve equals zero in any direction. To calculate this point the derivative of the implicit equation is parameterized:

$$\nabla_{xy}=\langle 3ax^2-6bxy+3cy^2+6ex-3fy+3h,\ -3bx^2+6cxy-3dy^2-3fx+6gy-3i\rangle \tag{32}$$

This derivative is substituted by the parametric equation of

$$x=-\hat{t}^3x_a+3\hat{t}^2x_b-3\hat{t}x_c+x_d \quad \text{and} \quad y=-\hat{t}^3y_a+3\hat{t}^2y_b-3\hat{t}y_c+y_d.$$

The two common roots of these two derivatives are the parameter t_1 and t_2 of the self intersection point. This seems to be a polynomial of sixth degree to solve but fortunately the two highest degrees vanish in this case.

The self intersection point of the cubic Bézier curve is calculated by the two common roots of the two polynomials:

$$\hat{t}^4x_a c_{ab}^2-\hat{t}^3c_{ab}(x_a c_{ac}+2x_b c_{ab})+\hat{t}^2(x_a c_{ac}^2-x_a c_{ab}c_{bc}+3x_c c_{ab}^2)+$$
$$+\hat{t}(6x_b c_{ab}c_{bc}-3x_b c_{ac}^2+x_a c_{ac}c_{bc})+x_c(c_{ac}^2-3c_{ab}c_{bc})=0,$$
$$\hat{t}^4y_a c_{ab}^2-\hat{t}^3c_{ab}(y_a c_{ac}+2y_b c_{ab})+\hat{t}^2(y_a c_{ac}^2-y_a c_{ab}c_{bc}+3y_c c_{ab}^2)+$$
$$+\hat{t}(6y_b c_{ab}c_{bc}-3y_b c_{ac}^2+y_a c_{ac}c_{bc})+y_c(c_{ac}^2-3c_{ab}c_{bc})=0.$$

Newton-Raphson iteration can be used to calculate the roots of polynomials. But to find the common roots of two polynomials the Bezout's resultant helps again to obtain an analytic solution.

Given two polynomials $f(t)=\sum a_i t^i$, $g(t)=\sum b_i t^i$ the homogeneous linear equation of Bezout's resultant of the fourth degree makes:

$$\begin{bmatrix} d_{01} & d_{02} & d_{03} & d_{04} \\ d_{02} & d_{03}+d_{12} & d_{04}+d_{13} & d_{14} \\ d_{03} & d_{04}+d_{13} & d_{14}+d_{23} & d_{24} \\ d_{04} & d_{14} & d_{24} & d_{34} \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} = 0 \quad \text{where} \quad d_{ij}=a_i b_j - a_j b_i. \tag{33}$$

The polynomials $f(t)$ and $g(t)$ have a common root if and only if the determinant of R(f,g) equals zero. These roots can be found by performing the Gaussian elimination on the rows of R(f,g). After elimination if the last non-zero row is $(0, ..., 0, h_0, h_1, ..., h_k)$, then the common roots of $f(t)$, $g(t)$ are simply the roots of the polynomial $h(t)=\sum h_i t^i$. [Goldman et al., 1985]

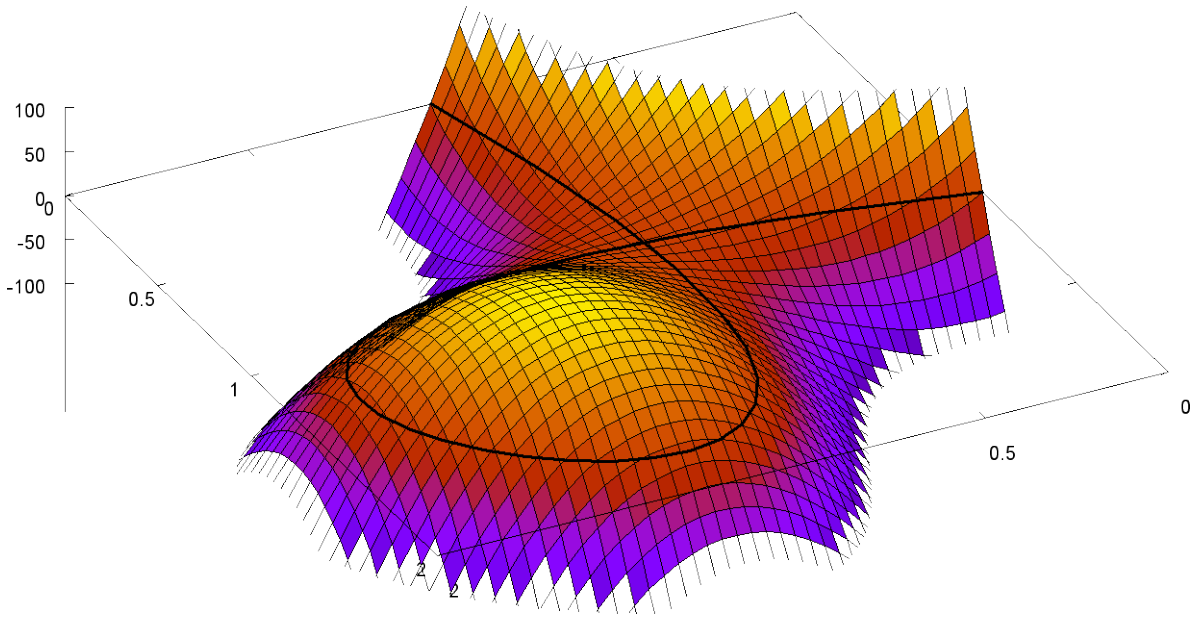


Figure 19: 3D surface plot of a cubic Bézier curve with self-intersection

The matrix coefficients of Bezout's resultant of the cubic Bézier curve make:

$$d_{01}=2c_{bc}(3c_{ab}c_{bc}-c_{ac}^2)^2,\quad d_{02}=-c_{ac}(c_{ab}c_{bc}-c_{ac}^2)(3c_{ab}c_{bc}-c_{ac}^2),$$
$$d_{03}=-c_{ab}(2c_{ab}c_{bc}+c_{ac}^2)(3c_{ab}c_{bc}-c_{ac}^2),\quad d_{04}=c_{ab}^2c_{ac}(3c_{ab}c_{bc}-c_{ac}^2),$$
$$d_{12}=-3c_{ab}(5c_{ab}c_{ac}^2c_{bc}-8c_{ab}^2c_{bc}^2-c_{ac}^4),\quad d_{13}=c_{ab}^2c_{ac}(4c_{ab}c_{bc}-3c_{ac}^2),$$
$$d_{14}=-3c_{ab}^3(2c_{ab}c_{bc}-c_{ac}^2),\quad d_{23}=c_{ab}^3(8c_{ab}c_{bc}+c_{ac}^2),\quad d_{24}=-3c_{ab}^4c_{ac},\quad d_{34}=2c_{ab}^5.$$

These values only depend on c_{ab}, c_{ac} and c_{bc}. The determinant of this resultant matrix is always zero.

Eliminating one row and column of the matrix gets the quadratic equation to calculate the roots of the self-intersection point. Taking row 2 and 3 the parameter of the self-intersection point makes: $\hat{t}_{1,2}=\frac{c_{ac}\pm\sqrt{12\,c_{ab}c_{bc}-3\,c_{ac}^2}}{2\,c_{ab}}$. (34)

Please note that $\hat{t}_1$ and $\hat{t}_2$ also do not depend on x_d, y_d, which are like the center of the Bézier curve.

For $4\,c_{ab}c_{bc}<c_{ac}^2$ the cubic implicit equation has a complex self-intersection point (acnode). If $4\,c_{ab}c_{bc}=c_{ac}^2$ the curve has a cusp. If $|\hat{t}_{1,2}|<½$ then the cubic Bézier segment has a self-intersection point (crunode). Otherwise only the implicit equation outside the parameter interval has a self-intersection.

5.6 Gradient at P_0

The gradient of the curve at P_0 is either positive or negative in plot direction. Since the algorithm depends on a continuously positive (or negative) slope the values have to be negated in case of a negative gradient to be able to use the same algorithm. How to calculate the gradient at P_0?

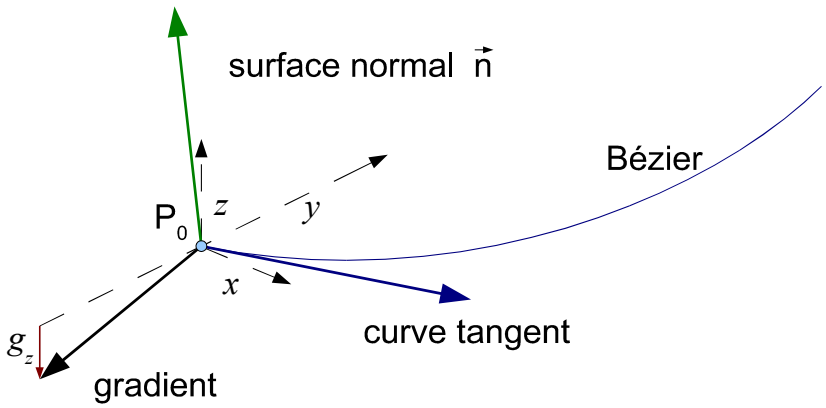


Figure 20: Curve gradient

The gradient is the cross product of the surface normal and the curve tangent vector.

The normal to the surface of the equation e(*x*,*y*) = *z* in figure (20) makes

$\vec{n}=\left[-\frac{\partial \mathrm{e}(x,y)}{\partial x},-\frac{\partial \mathrm{e}(x,y)}{\partial y},1\right]$.

With the definition of the grade $g=\frac{27}{4}\left((c_{ab}+c_{ac})^2-c_{ab}(c_{ac}+3\,c_{bc})\right)$ the derivatives of the equation make $\frac{\partial \mathrm{e}(x_0,y_0)}{\partial x}=g\,(y_0-y_1)$ and $\frac{\partial \mathrm{e}(x_0,y_0)}{\partial y}=-g\,(x_0-x_1)$. The gradient at

P_0 makes then

$$\nabla e(x_0, y_0) = \vec{n} \times [x_1 - x_0, y_1 - y_0, 0] = [y_0 - y_1, x_1 - x_0, g((x_0 - x_1)^2 + (y_0 - y_1)^2)].$$

The z component (g_z in figure 20) of the gradient (proportional to g) is only negative if P_0 is part of the self intersection loop. If the grade g equals zero the point P_0 itself is the crunode (or cusp).

This information can be used to take special care since drawing the self-intersection loop always includes the danger that another part of the curve comes too close for the algorithm to work correctly.

5.7 Inflection point

The position of the inflection points becomes important if the cubic Bézier is approximated by sets of connected quadratic Bézier segments. A cubic Bézier curve changes its bending direction at the inflection point.

The inflection point is the point where the curvature of a curve equals zero. The curvature of a curve is calculated by

$$\kappa = \frac{1}{r} = \frac{d\varphi}{ds} = \frac{|\dot{B}(t) \times \ddot{B}(t)|}{|\dot{B}(t)|^3} = \frac{\dot{f}_x(t)\ddot{f}_y(t) - \dot{f}_y(t)\ddot{f}_x(t)}{\sqrt{\left(\dot{f}_x^2(t) + \dot{f}_y^2(t)\right)^3}}. \quad (35)$$

In case of the cubic Bézier curve the inflection point could be determined by the quadratic equation of

$$\hat{t}^2 c_{ab} - \hat{t}\, c_{ac} + c_{bc} = 0. \quad (36)$$

5.8 Cubic troubles

Again the algorithm runs into troubles for certain Bézier curves. Figure 21 shows a curve with ambitious error values at P_0. The situation becomes clear if the entire curve is plotted. The point at P_0 is near a self intersection point.

The algorithm has a real dilemma with flat self-intersection loops. It cannot plot from the intersection side since the error values change sign there. If the algorithm wants to plot from the flat tip side there is the danger that the other part of the curve is too close and the error values are too confusing. The algorithm fails if the curve contains a flat self-intersection loop.

Figure 21 shows a curve where the loop is very flat. In that case both ends could be acute angles and lead into narrow or same pixel. The algorithm can not plot a curve if both ends start with such confusing error values. Two possibilities could solve the problem. The first would be the same used for quadratic rational Béziers. The curve is additionally subdivided at the broadest position of the loop and the plot starts there. The curves get a bit edgy sometimes by this solution. The other one is to use a higher resolution raster for the plot. The resolution is selected by the length of the legs. This method slightly decreases the plot speed but leads to smoother curves.

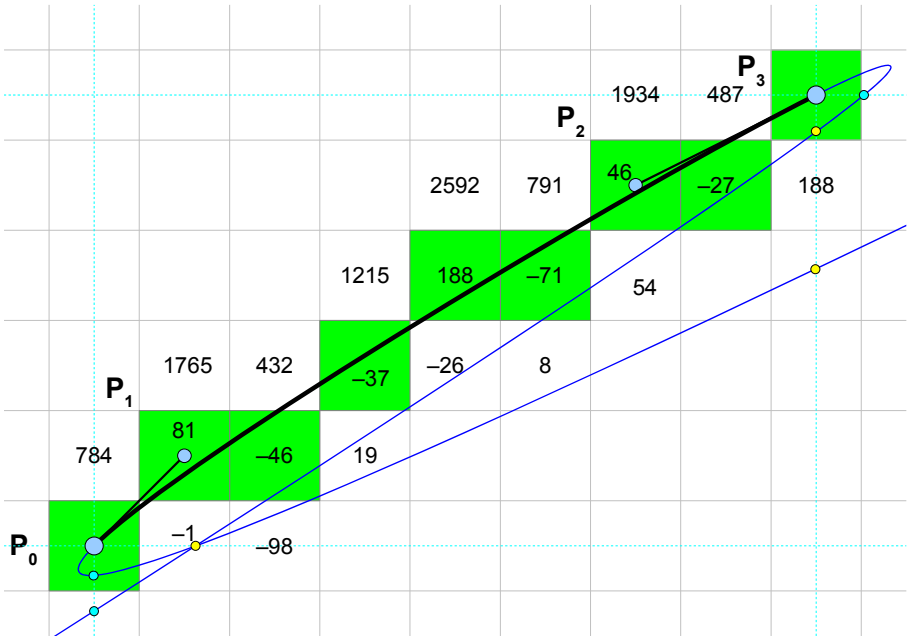


Figure 21: Cubic Bézier with confusing error values

5.9 Cubic algorithm

The basic algorithm needs a continuously rising or falling curve. It is not even simple to include a check of the parameters to ensure this condition. The first is that the lines P_1-P_0 and P_3-P_2 are both either rising or falling. But the slope of P_2-P_1 could be negated to the others and the curve could still be plot. So the second condition must be that either the line P_3-P_0 has the same slope than P_2-P_1 or the first derivative of the parametric equation has real roots.

The assert statement must also consider certain rounding errors. If the parameter is of single floating point type with 23 bit mantissa and the parameter of the coordinates don't exceed 16 bit then epsilon must be greater than 2^{16-23}.

For simple (and faster) code the calculation of d_x and d_{xx} is one y-step ahead. The calculation of d_y and d_{yy} is one x-step ahead. The calculation of the error value is one x- and one y-step ahead.

The cubic Bézier curve could have a cusp or self-intersection loop. Even if the loop is not directly visible with the specific parameter the entire curve could have a loop and the visible segment be a part of it. This cusp or loop causes many troubles. The algorithm can not plot through the self-intersection. Therefore the plot starts from both ends. If the error values get confused because the gradient changes its sign then the other end is tried. Consequently the function is able to handle Bézier curves with self-intersection points.

Another problem arises if this loop is very flat like in figure 21. Then one end of the curve is prossibly blocked by the self-intersection point and on the other end has too confusing

error values for the algorithm. The resolution of the pixel is increased in that case. The algorithm uses the value of the grade to check if the resolution factor has to be increased.

The algorithm also looks one pixel ahead to detect a cusp of self-intersection ($d_x > 0$ or $d_y < 0$). But at the start point these values could still be negated for one pixel ahead although they are not for the current. The algorithm must therefore check the values at the current pixel ($d_x > d_{xy}$ or $d_y < d_{xy}$) and switch to the check of one pixel ahead only if the values there became valid.

```
void plotCubicBezierSeg(int x0, int y0, float x1, float y1,
                        float x2, float y2, int x3, int y3)
{                                       /* plot limited cubic Bezier segment */
   int f, fx, fy, leg = 1;
   int sx = x0 < x3 ? 1 : -1, sy = y0 < y3 ? 1 : -1;      /* step direction */
   float xc = -fabs(x0+x1-x2-x3), xa = xc-4*sx*(x1-x2), xb = sx*(x0-x1-x2+x3);
   float yc = -fabs(y0+y1-y2-y3), ya = yc-4*sy*(y1-y2), yb = sy*(y0-y1-y2+y3);
   double ab, ac, bc, cb, xx, xy, yy, dx, dy, ex, *pxy, EP = 0.01;

                                        /* check for curve restrains */
   /* slope P0-P1 == P2-P3   and  (P0-P3 == P1-P2      or  no slope change) */
   assert((x1-x0)*(x2-x3) < EP && ((x3-x0)*(x1-x2) < EP || xb*xb < xa*xc+EP));
   assert((y1-y0)*(y2-y3) < EP && ((y3-y0)*(y1-y2) < EP || yb*yb < ya*yc+EP));

   if (xa == 0 && ya == 0) {                            /* quadratic Bezier */
      sx = floor((3*x1-x0+1)/2); sy = floor((3*y1-y0+1)/2); /* new midpoint */
      return plotQuadBezierSeg(x0,y0, sx,sy, x3,y3);
   }
   x1 = (x1-x0)*(x1-x0)+(y1-y0)*(y1-y0)+1;                 /* line lengths */
   x2 = (x2-x3)*(x2-x3)+(y2-y3)*(y2-y3)+1;
   do {                                                /* loop over both ends */
      ab = xa*yb-xb*ya; ac = xa*yc-xc*ya; bc = xb*yc-xc*yb;
      ex = ab*(ab+ac-3*bc)+ac*ac;     /* P0 part of self-intersection loop? */
      f = ex > 0 ? 1 : sqrt(1+1024/x1);             /* calculate resolution */
      ab *= f; ac *= f; bc *= f; ex *= f*f;          /* increase resolution */
      xy = 9*(ab+ac+bc)/8; cb = 8*(xa-ya);/* init differences of 1st degree */
      dx = 27*(8*ab*(yb*yb-ya*yc)+ex*(ya+2*yb+yc))/64-ya*ya*(xy-ya);
      dy = 27*(8*ab*(xb*xb-xa*xc)-ex*(xa+2*xb+xc))/64-xa*xa*(xy+xa);
                                       /* init differences of 2nd degree */
      xx = 3*(3*ab*(3*yb*yb-ya*ya-2*ya*yc)-ya*(3*ac*(ya+yb)+ya*cb))/4;
      yy = 3*(3*ab*(3*xb*xb-xa*xa-2*xa*xc)-xa*(3*ac*(xa+xb)+xa*cb))/4;
      xy = xa*ya*(6*ab+6*ac-3*bc+cb); ac = ya*ya; cb = xa*xa;
      xy = 3*(xy+9*f*(cb*yb*yc-xb*xc*ac)-18*xb*yb*ab)/8;

      if (ex < 0) {       /* negate values if inside self-intersection loop */
         dx = -dx; dy = -dy; xx = -xx; yy = -yy; xy = -xy; ac = -ac; cb = -cb;
      }                                  /* init differences of 3rd degree */
      ab = 6*ya*ac; ac = -6*xa*ac; bc = 6*ya*cb; cb = -6*xa*cb;
      dx += xy; ex = dx+dy; dy += xy;                 /* error of 1st step */

```

```
      for (pxy = &xy, fx = fy = f; x0 != x3 && y0 != y3; ) {
         setPixel(x0,y0);                                    /* plot curve */
         do {                                  /* move sub-steps of one pixel */
            if (dx > *pxy || dy < *pxy) goto exit;      /* confusing values */
            y1 = 2*ex-dy;                   /* save value for test of y step */
            if (2*ex >= dx) {                                   /* x sub-step */
               fx--; ex += dx += xx; dy += xy += ac; yy += bc; xx += ab;
            }
            if (y1 <= 0) {                                      /* y sub-step */
               fy--; ex += dy += yy; dx += xy += bc; xx += ac; yy += cb;
            }
         } while (fx > 0 && fy > 0);                      /* pixel complete? */
         if (2*fx <= f) { x0 += sx; fx += f; }                    /* x step */
         if (2*fy <= f) { y0 += sy; fy += f; }                    /* y step */
         if (pxy == &xy && dx < 0 && dy > 0) pxy = &EP;/* pixel ahead valid */
      }
exit: xx = x0; x0 = x3; x3 = xx; sx = -sx; xb = -xb;           /* swap legs */
      yy = y0; y0 = y3; y3 = yy; sy = -sy; yb = -yb; x1 = x2;
   } while (leg--);                                        /* try other end */
   plotLine(x0,y0, x3,y3);     /* remaining part in case of cusp or crunode */
}
```

Listing 14: Plotting a cubic Bézier segment

The algorithm tries to plot from either end of the curve and stops if the error values get confused. This happens in case of a self-intersection and especially of a cusp. The curve is then finished by a line.

It would be possible to test the break condition of the pixel loop just for the end point. But if the end point is missed due to a slight rounding error the algorithm wouldn't stop. The implementation in listing 14 ensures that the curve is not plot beyond the end point.

A few further optimizations are possible like writing a separate loop for curves without loop (self intersection). This would reduce the overhead in the pixel loop for most the drawn Bézier curves. A reference could be used instead of the pointer variable if applicable.

The algorithm could be simplified if no cubic Bézier curves with a loop or cusp have to be rasterized. The variation for increased resolution could be omitted in that case.

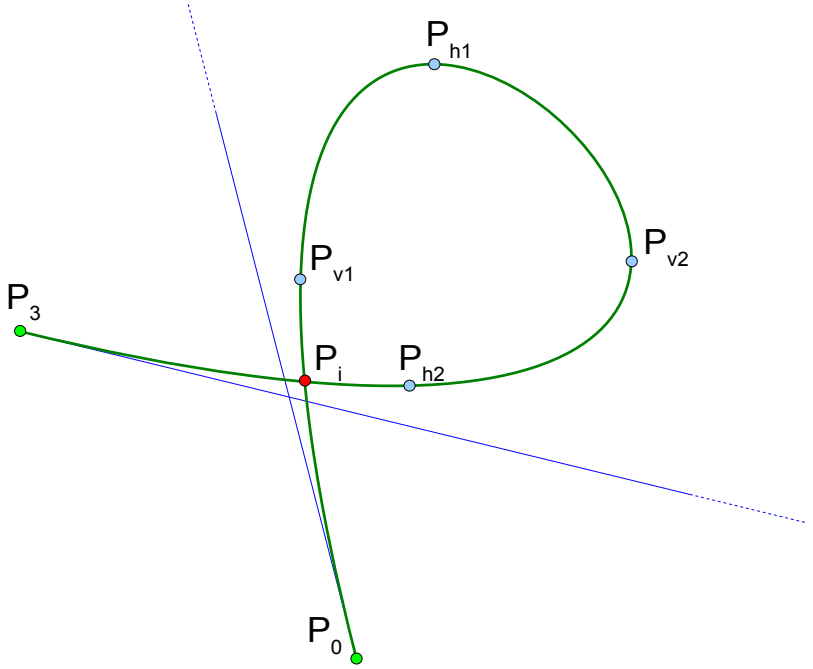


Figure 22: Cubic Bézier curve divided into segments

5.10 Subdividing cubic Béziers

A complex cubic Bézier curve must be cut at horizontal and vertical maximum and minimum. This is accomplished by the first parametric derivative which makes the control points $D_i = n(P_{i+1}–P_i)$.

The parametric value *t* of these points is calculated by the roots of the first parametric derivative: $\frac{\mathrm{d}x}{\mathrm{d}\hat{t}} = -3(\hat{t}^2 x_a - 2\hat{t}\,x_b + x_c) = 0$ and $\frac{\mathrm{d}y}{\mathrm{d}\hat{t}} = -3(\hat{t}^2 y_a - 2\hat{t}\,y_b + y_c) = 0.$

Each quadratic equation has none, one or two real roots. At these points P_{h1}, P_{h2}, P_{v1} and P_{v2} of the parameter t the curve must be subdivided by *de Casteljau's* algorithm.

The Bézier curve is reparametrized by the new range from t_1 to t_2:

$$\hat{t} = \tilde{t}\,(t_2 - t_1) + t_2 + t_1 \quad [t_1 \le \hat{t} \le t_2]\ [-½ \le \tilde{t} \le ½].$$

If the segment between t_1 and t_2 should be plot the new points of the Bézier curve make:

$$\tilde{x}_0 = -t_1^3 x_a + 3t_1^2 x_b - 3t_1 x_c + x_d, \quad \tilde{x}_1 = -t_2(t_1^2 x_a - 2t_1 x_b + x_c) + t_1^2 x_b - 2t_1 x_c + x_d,$$
$$\tilde{x}_2 = -t_1(t_2^2 x_a - 2t_2 x_b + x_c) + t_2^2 x_b - 2t_2 x_c + x_d \quad \text{and} \quad \tilde{x}_3 = -t_2^3 x_a + 3t_2^2 x_b - 3t_2 x_c + x_d. \quad (37)$$

Substitution of the new parameter in the cubic Bézier equation

$$x = -\tilde{t}^3 \tilde{x}_a + 3\tilde{t}^2 \tilde{x}_b - 3\tilde{t}\,\tilde{x}_c + \tilde{x}_d$$

(plus the appropriate equation for *y*) and comparing the terms makes:

$$\tilde{x}_a = (t_2 - t_1)^3 x_a, \quad \tilde{x}_b = (t_2 - t_1)^2 (2x_b - (t_1 + t_2) x_a),$$

$$\tilde{x}_c=(t_2-t_1)((t_1+t_2)^2 x_a-4(t_1+t_2)x_b+4x_c) \quad \text{and}$$
$$\tilde{x}_d=-(t_1+t_2)^3 x_a+6(t_1+t_2)^2 x_b-12(t_1+t_2)x_c+8x_d. \tag{38}$$

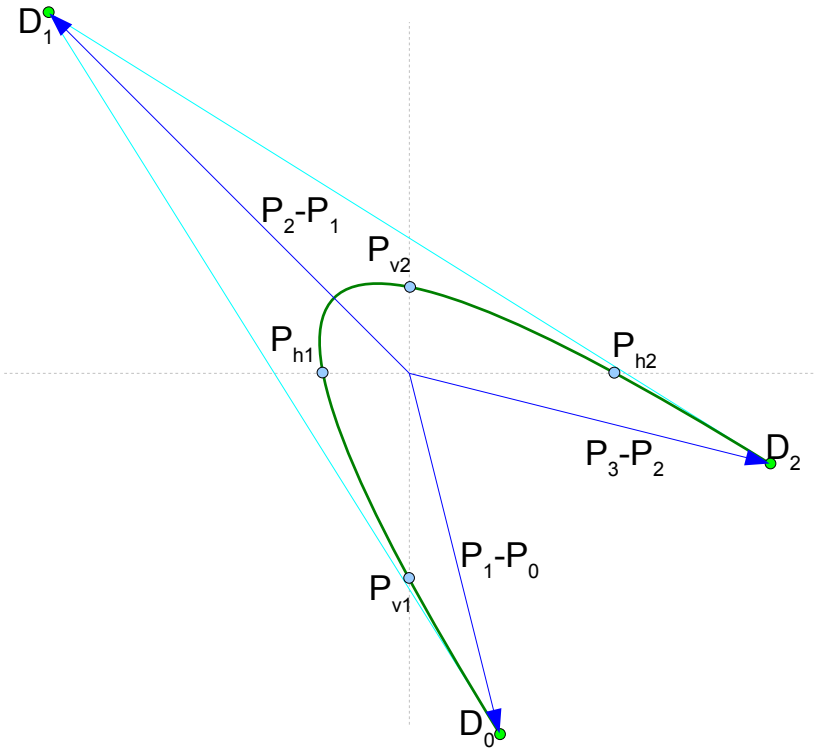


Figure 23: Horizontal and vertical roots of a cubic Bézier

5.11 Drawing any cubic Bézier curve

The Bézier curve is subdivided at horizontal and vertical gradient changes. These points are then sorted to get consecutive curve segments. Since only four points at most have to be sorted the simplest sorting algorithm is efficiently enough. The curve is not subdivided at self-intersection point since the sub-segment plot function is able to handle this case.

Another problem would arise when the intermediate points P_1 or P_2 are rounded to integers due to the subdivision. It could happen that the carefully avoided situation of gradient change suddenly reappears again due to the rounding. To avoid a significant change only the boundary of the subdivided curve is therefore scaled to integers. For this reason the intermediate points of the sub-segment plot function must be of floating point type.

It is essential that the computation of the parameter t is of double accuracy especially the square root function. Insufficient accuracy may change the parameter of the subdivided curve slightly so that it does not match the necessary restrains anymore and the assert statement in the sub-segment function would fail.

```
void plotCubicBezier(int x0, int y0, int x1, int y1,
                     int x2, int y2, int x3, int y3)
{                                       /* plot any cubic Bezier curve */
   int n = 0, i = 0;
   long xc = x0+x1-x2-x3, xa = xc-4*(x1-x2);
   long xb = x0-x1-x2+x3, xd = xb+4*(x1+x2);
   long yc = y0+y1-y2-y3, ya = yc-4*(y1-y2);
   long yb = y0-y1-y2+y3, yd = yb+4*(y1+y2);
   float fx0 = x0, fx1, fx2, fx3, fy0 = y0, fy1, fy2, fy3;
   double t1 = xb*xb-xa*xc, t2, t[5];
                           /* sub-divide curve at gradient sign changes */
   if (xa == 0) {                                          /* horizontal */
      if (abs(xc) < 2*abs(xb)) t[n++] = xc/(2.0*xb);     /* one change */
   } else if (t1 > 0.0) {                                /* two changes */
      t2 = sqrt(t1);
      t1 = (xb-t2)/xa; if (fabs(t1) < 1.0) t[n++] = t1;
      t1 = (xb+t2)/xa; if (fabs(t1) < 1.0) t[n++] = t1;
   }
   t1 = yb*yb-ya*yc;
   if (ya == 0) {                                            /* vertical */
      if (abs(yc) < 2*abs(yb)) t[n++] = yc/(2.0*yb);     /* one change */
   } else if (t1 > 0.0) {                                /* two changes */
      t2 = sqrt(t1);
      t1 = (yb-t2)/ya; if (fabs(t1) < 1.0) t[n++] = t1;
      t1 = (yb+t2)/ya; if (fabs(t1) < 1.0) t[n++] = t1;
   }
   for (i = 1; i < n; i++)                    /* bubble sort of 4 points */
      if ((t1 = t[i-1]) > t[i]) { t[i-1] = t[i]; t[i] = t1; i = 0; }

   t1 = -1.0; t[n] = 1.0;                            /* begin / end point */
   for (i = 0; i <= n; i++) {           /* plot each segment separately */
      t2 = t[i];                          /* sub-divide at t[i-1], t[i] */
      fx1 = (t1*(t1*xb-2*xc)-t2*(t1*(t1*xa-2*xb)+xc)+xd)/8-fx0;
      fy1 = (t1*(t1*yb-2*yc)-t2*(t1*(t1*ya-2*yb)+yc)+yd)/8-fy0;
      fx2 = (t2*(t2*xb-2*xc)-t1*(t2*(t2*xa-2*xb)+xc)+xd)/8-fx0;
      fy2 = (t2*(t2*yb-2*yc)-t1*(t2*(t2*ya-2*yb)+yc)+yd)/8-fy0;
      fx0 -= fx3 = (t2*(t2*(3*xb-t2*xa)-3*xc)+xd)/8;
      fy0 -= fy3 = (t2*(t2*(3*yb-t2*ya)-3*yc)+yd)/8;
      x3 = floor(fx3+0.5); y3 = floor(fy3+0.5); /* scale bounds to int */
      if (fx0 != 0.0) { fx1 *= fx0 = (x0-x3)/fx0; fx2 *= fx0; }
      if (fy0 != 0.0) { fy1 *= fy0 = (y0-y3)/fy0; fy2 *= fy0; }
      if (x0 != x3 || y0 != y3)                    /* segment t1 - t2 */
         plotCubicBezierSeg(x0,y0, x0+fx1,y0+fy1, x0+fx2,y0+fy2, x3,y3);
      x0 = x3; y0 = y3; fx0 = fx3; fy0 = fy3; t1 = t2;
   }
}
```

Listing 15: Sub-dividing a cubic Bézier curve

The middle control points P_1 and P_2 could be changed to the thru points $\mathring{P}_1$, $\mathring{P}_2$ by

$$P_1 = \frac{-5P_0 + 18\mathring{P}_1 - 9\mathring{P}_2 + 2P_3}{6} \quad \text{and} \quad P_2 = \frac{2P_0 - 9\mathring{P}_1 + 18\mathring{P}_2 - 5P_3}{6}. \tag{39}$$

6 Rational cubic Béziers

The parametric equation of the rational cubic Bézier makes [Marsh, 2005, p. 175]:

$$\mathrm{B}_3(t)=\frac{(1-t)^3 w_0 \mathrm{P}_0+3(1-t)^2 t\, w_1 \mathrm{P}_1+3(1-t) t^2 w_2 \mathrm{P}_2+t^3 w_3 \mathrm{P}_3}{(1-t)^3 w_0+3(1-t)^2 t\, w_1+3(1-t) t^2 w_2+t^3 w_3} \tag{40}$$

The weights of the implicit equation of the rational quadratic Bézier could be normalized to simplify the calculations without changing the curve. The weights of the end points P_0 and P_2 were distributed to the others. The same is possible for any rational Bézier of degree n. By comparison of the implicit equation with normalized and not normalized weights the weights of the rational Bézier can be normalized by the substitution of

$$\hat{w}_i=\frac{w_i}{\sqrt[n]{w_0^{n-i} w_n^i}}. \tag{41}$$

The weights of the end points P_0 and P_n became now $\hat{w}_0=\hat{w}_n=1$.

For the cubic Bézier the normalized weights make $\hat{w}_1=\frac{w_1}{\sqrt[3]{w_0^2 w_3}}$ and $\hat{w}_2=\frac{w_2}{\sqrt[3]{w_0 w_3^2}}$.

The implicit equation of the rational cubic Bézier is needed to apply the same algorithm as done for the previous curves. This equation gets very complex.

Using resultants the third degree of the rational cubic Bézier curve is implicitized the same way as equation (25) for the non-rational curve:

$$\mathrm{f}(x,y)=\begin{vmatrix} d_{01} & d_{02} & d_{03} \\ d_{02} & d_{03}+d_{12} & d_{13} \\ d_{03} & d_{13} & d_{23} \end{vmatrix}=0 \text{ where } d_{ij}=w_i w_j \binom{3}{i}\binom{3}{j} \begin{vmatrix} x & y & 1 \\ x_i & y_i & 1 \\ x_j & y_j & 1 \end{vmatrix} \tag{42}$$

No simplification could be found to make the computation of the initial values easier as was for the normal cubic Bézier curve. An algorithm with the same principle is possible but too complex for this work. Instead algorithms are worked out to subdivide the curve and draw the rational quadratic segments.

6.1 Rational degree reduction

The same consideration as chapter 5.1 for non-rational cubic Béziers could be made to approximate rational Bézier curves by quadratic ones.

To keep the deviation small it is again advisable to take the subdivision in three steps:

- the curve is subdivided at horizontal and vertical stationary points (chapter 6.3)
- the curve is subdivided at the inflection point (chapter 6.4)
- the curve is subdivided in two rational quadratic Béziers

The following considerations keep the deviations small. The cubic Bézier P_0-P_1-P_2-P_3 in figure 17 of chapter 5 is exactly halved (at t = ½) by DeCasteljou subdivision. The subdivi-

sion point P_c remains exactly at the cubic Bézier curve. The intermediate points in figure 17 of the rational cubic curve are then calculated by

$$w_a = w_1, \quad P_a = \frac{3 w_1 P_1 + w_0 P_0}{3 w_1 + w_0}, \quad w_b = w_2, \quad P_b = \frac{3 w_2 P_2 + w_3 P_3}{3 w_2 + w_3},$$

$$P_c = \frac{w_0 P_0 + 3 w_1 P_1 + 3 w_2 P_2 + w_3 P_3}{w_0 + 3 w_1 + 3 w_2 + w_3}.$$

The two subdivided rational Bézier curves can be drawn by the algorithm of chapter 4.

6.2 Sub-dividing rational cubic Béziers

The cubic curve must be subdivided to simplify the drawing of the curve. The subdivision is done by finding an appropriate value of the parameter t (for example stationary points). The position and the weight of the two subdivided curves have to be found. The curve in figure 24 is subdivided at point P_c into the two curves P_0-P_a-P_b-P_c and P_c-P_d-P_e-P_3.

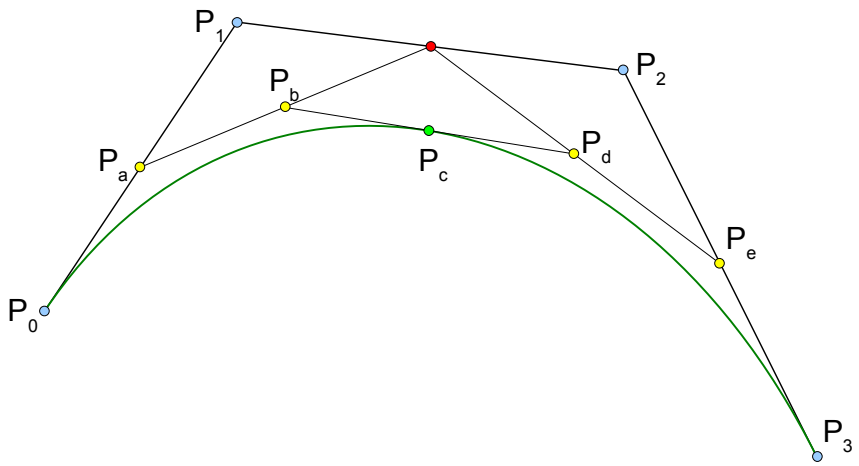


Figure 24: Subdividing a rational cubic Bézier

The *de Casteljau* algorithm is extended for rational Béziers. The additional points are now calculated by the following equations:

$$w_a P_a = (1-t) w_0 P_0 + t w_1 P_1, \qquad w_a = (1-t) w_0 + t w_1,$$

$$w_b P_b = (1-t)^2 w_0 P_0 + 2(1-t) t w_1 P_1 + t^2 w_2 P_2, \qquad w_b = (1-t)^2 w_0 + 2(1-t) t w_1 + t^2 w_2,$$

$$w_c P_c = (1-t)^3 w_0 P_0 + 3(1-t)^2 t w_1 P_1 + 3(1-t) t^2 w_2 P_2 + t^3 w_3 P_3,$$

$$w_c = (1-t)^3 w_0 + 3(1-t)^2 t w_1 + 3(1-t) t^2 w_2 + t^3 w_3,$$

$$w_d P_d = (1-t)^2 w_1 P_1 + 2(1-t) t w_2 P_2 + t^2 w_3 P_3, \qquad w_d = (1-t)^2 w_1 + 2(1-t) t w_2 + t^2 w_3,$$

$$w_e P_e = (1-t) w_2 P_2 + t w_3 P_3, \qquad w_e = (1-t) w_2 + t w_3.$$

By the use of equation (41) it is possible to make the weight of point P_c equal to one by adapting the weights of the other points.

6.3 Root finding

The rational cubic Bézier must be subdivided at the stationary points. These points are the maximums and minimums of the curve in x- and y-direction. These points can be calculated by setting the first derivation of the parametric equation (40) to zero. This equation becomes a bit simpler by introducing the substitution of $\mathrm{P}_{ij}=(w_i-w_j)\mathrm{P}_i\mathrm{P}_j$.

$$(\mathrm{P}_{01}-2\mathrm{P}_{02}+\mathrm{P}_{03}+3\mathrm{P}_{12}-2\mathrm{P}_{13}+\mathrm{P}_{23})t^4-2(2\mathrm{P}_{01}-3\mathrm{P}_{02}+\mathrm{P}_{03}+3\mathrm{P}_{12}-3\mathrm{P}_{13})t^3+$$
$$+(6\mathrm{P}_{01}-6\mathrm{P}_{02}+\mathrm{P}_{03}+3\mathrm{P}_{12})t^2-2(2\mathrm{P}_{01}-\mathrm{P}_{02})t+\mathrm{P}_{01}=0.$$

This equation is a polynomial of the fourth degree. The analytical calculation of the roots of the fourth degree is possible but difficult. The following solution tries to find an algorithm as simple as possible.

Only the real roots are of interest for this application. If the highest coefficient of the following polynomial equations is zero the equation could be reduced by one degree. The coefficient is therefore by assumed to be unequal to zero.

The accuracy of the calculation needs special attention. The numbers can get quite large due to the powers. Rounding errors occur if such large numbers are added or subtracted due to the limited accuracy of the calculation.

The implementations are as simple as possible and use a minimum of transcendental functions.

6.3.1 Quadratic equation

For the quadratic polynomial equation $a_2x^2+a_1x+a_0=0$ the substitution of $p=-\frac{a_1}{2a_2},\ q=\frac{a_0}{a_2}$ makes the calculation easier.

The number of real roots depends on the discriminant shown in the following table:

Discriminant	Number of real roots	Root value(s)
$p^2 < q$	0	-
$p^2 = q$	1	$x = p$
$p^2 > q$	2	$x_1 = p+\sqrt{p^2-q}$ $x_2 = q/x_1$

The calculation of x_2 by the division is more stable than the negative root.

6.3.2 Cubic equation

For the cubic polynomial equation $a_3x^3+a_2x^2+a_1x+a_0=0$ the substitution of $p=-\frac{a_2}{3a_3},\ q=p^2-\frac{a_1}{3a_3},\ r=\frac{3q-p^2}{2}p+\frac{a_0}{2a_3}$ makes the calculation again easier.

These definitions also show the degree relations between the coefficients which is relevant to avoid an underflow of floating point numbers.

The substitution of $x = z + p$ gets the depressed cubic $z^3 - 3qz + 2r = 0.$ By Vieta's substitution of $z = y + q/y$ this equation could be turned into a quadratic equation in y^3: $y^6 + 2ry^3 + q^3 = 0.$ Back substitution gets the roots of the cubic equation.

The number of real roots depends on the discriminant shown in the following table:

Discriminant	Number of real roots	Root value(s)
$r = 0 \wedge q = 0$	1	$x = p$
$r^2 > q^3$	1	$y = \sqrt[3]{r + \sqrt{r^2 - q^3}}$ $x = p + y + q/y$
$r^2 = q^3$	2	$x_1 = p + r/q$ $x_2 = p - 2r/q$
$r^2 < q^3$	3	$y = \sqrt{q}\cos\left(\frac{1}{3}\arccos\frac{r}{\sqrt{q^3}}\right)$ $x_1 = p - 2y$ $x_{2,3} = p + y \pm \sqrt{3(q - y^2)}$

The last case uses the substitution of $z = 2\sqrt{q}\cos\theta$ for the depressed cubic to avoid the calculations of complex numbers. Dividing the equation by $2\sqrt{q^3}$ gets the cubic equation of $4\cos^3\theta - 3\cos\theta + \frac{r}{\sqrt{q^3}} = 0.$ Comparison with the equation of $\cos 3\theta = 4\cos^3\theta - 3\cos\theta$ gets the relation of $\cos 3\theta = \frac{-r}{\sqrt{q^3}}.$ The roots are then calculated by the entries of the table above.

6.3.3 Quartic equation

For the quartic polynomial equation $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0$ the substitution of $p = -\frac{a_3}{4a_4},\ q = 3p^2 - \frac{a_2}{2a_4},\ r = (q - p^2)p - \frac{a_1}{4a_4},\ s = \left((p^2 - 2q)p + 4r\right)p - \frac{a_0}{a_4}$ makes the calculation once again easier.

The substitution $x = z + p$ gets rid of the cubic term: $z^4 - 2qz^2 - 4rz - s = 0$

If r equals zero this is a quadratic equation in z^2. The original quartic is then solved by:

$$x = p \pm_1 \sqrt{q \pm_2 \sqrt{q^2 + s}} \qquad [r = 0]$$

If r does not equal zero the quartic equation can be turned to the solution of a resolvent cubic equation by several methods [Shmakov, 2011]. The algorithm of Euler solves the quartic equation by the resolvent cubic equation of:

$$4y^3 - 4qy^2 + (q^2 + s)y - r^2 = 0$$

This cubic equation can be solved by the method of the previous chapter. It always has a positive root.

If y is a positive root of the cubic equation then the roots of the original quartic make

$$x = p \pm_1 \sqrt{q - y \pm_2 r/\sqrt{y}} \pm_2 \sqrt{y} \qquad [r \neq 0]$$

The subscript of the plus/minus sign indicate the same sign of the four different solutions of the quartic equation. Only real roots are of interest for this application.

6.4 Rational inflection point

The calculation of the inflection point of the rational cubic Bézier is a bit more complicated than of the non-rational Bézier. Equation (35) is again used to get the curvature of the curve. The following substitution makes the computation easier:

$$c_{ijk} = w_i w_j w_k \begin{vmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_k & y_k & 1 \end{vmatrix}.$$

The inflection points of the rational cubic Bézier curve could now be determined by the cubic equation of

$$t^3(c_{012} - c_{013} + c_{023} - c_{123}) - t^2(3c_{012} - 2c_{013} + c_{023}) + t(3c_{012} - c_{013}) - c_{012} = 0. \qquad (43)$$

The roots of this equation could be solved using the method of the previous chapter.

7 Anti-aliasing

Rasterization always causes a staircase effect because of the limited resolution of an image. Anti-aliasing reduces this effect by setting the intensity values of the pixels.

The best anti-aliasing algorithm uses the distance of a pixel to the curve as information for the intensity. Pixel of a distance of zero get full intensity. If the center of a pixel is one pixel off the curve the pixel gets no intensity.

The distance d to the parametric curve $P(t) = (x(t), y(t))$ is calculated by the offset curve

$$P(t) \pm d \frac{\left(y'(t), -x'(t)\right)}{\sqrt{x'^2(t) + y'^2(t)}}. \tag{44}$$

This equation gets quite complicated. For example for the quadratic Bézier curve the offset curve is of degree six, for rational quadratic Bézier curve it is of degree eight. Too complicated for an easy drawing algorithm.

This algorithm can be extended to use anti-aliasing since the calculated error values represent the distance from the curve. Error values of zero mean at the line. By extending this error limit it is also possible to draw thicker lines or in combination thick anti-aliased lines.

The requirement for anti-aliasing is that a pixel should not only be lit or not but the intensity of the pixel depends on how far it is away from the curve. Pixel exactly at the curve get the highest intensity, pixel which are one unit away - the equivalent of the pixel size - are not lit at all. Pixel which are between zero and one unit away from the curve get an intensity according to their distance. This requirement also makes the intensity of a straight diagonal line equivalent to a vertical or horizontal line.

In the ideal anti-aliasing algorithm the curve is the crossing of a three dimensional landscape with hills and valleys. The curve along the coastline always has a slope of 45 degrees. The altitude of the landscape defines the intensity of the pixel.

The value of the error depends on the local gradient of curve. But fortunately the discrete value of the gradient is already calculated as the error difference in X and Y directions. So the maximum error value e_d for the edges of the line makes

$$e_d = \pm\sqrt{d_x^2 + d_y^2}. \tag{45}$$

The discrete value is an approximation of the gradient for curves of higher degrees than one but this slight approximation spares the complicated calculation of the gradient in such cases.

7.1 Anti-aliased line

The error value of the implicit equation defines the blending from the drawing color to the background color.

In opposite to listing 2 the condition for the pixel step is delayed until the curve is beyond the next pixel so that the algorithm can always set the current and the next pixel in X or Y direction according to the error value.

```
void plotLineAA(int x0, int y0, int x1, int y1)
{     /* draw a black (0) anti-aliased line on white (255) background */
   int dx =  abs(x1-x0), sx = x0 < x1 ? 1 : -1;
   int dy = -abs(y1-y0), sy = y0 < y1 ? 1 : -1;
   int err = dx+dy, e2, x2;                          /* error value e_xy */
   float ed = dx-dy == 0 ? 1 : sqrt((float)dx*dx+(float)dy*dy);

   for ( ; ; ){                                                /* pixel loop */
      setPixelAA(x0,y0, 255*abs(err-dx-dy)/ed);
      e2 = err; x2 = x0;
      if (2*e2+dx >= 0) {                                       /* x step */
         if (x0 == x1) break;
         if (e2-dy < ed) setPixelAA(x0,y0+sy, 255*(e2-dy)/ed);
         err += dy; x0 += sx;
      }
      if (2*e2+dy <= 0) {                                       /* y step */
         if (y0 == y1) break;
         if (dx-e2 < ed) setPixelAA(x2+sx,y0, 255*(dx-e2)/ed);
         err += dx; y0 += sy;
      }
   }
}
```

Listing 16: Program to plot an anti-aliased line

The conditional setting of the pixel color is necessary because some-times the pixel is too far away from the curve (more than one pixel) and doesn't need to be set.

The divisions in the pixel loop for the color calculation could be replaced by multiplications and shift operations for speed reasons.

Some further optimizations are possible. For instance two different loops for line slopes below and above one would the if condition move out of the loop and therefore speed up the algorithm. Another optimization would be to start stetting the pixel from both ends simultaneously towards the middle since the line is always symmetric.

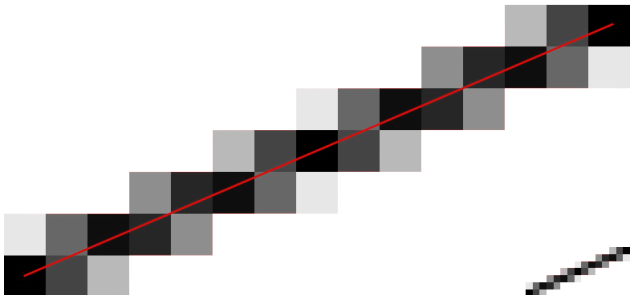


Figure 25: Example of an anti-aliased line

Setting the pixel intensity directly only works if the pixels don't overlap with other drawings. Otherwise the pixel setting must be replaced by a blending function:

```
setPixelAA(x,y,(blend*getPixelAA(x,y)+(255-blend)*lineColor)/255).
```

The variable *blend* is the intensity value of the previous listing. A value of zero means the line color, a value of 255 the background color. Other blending functions like minimum or maximum are also possible. For color images this function must be used for every color channel separately.

The next section (7.3) introduces a possible approximation for the square root function for further improvement.

The algorithm works a bit different than Xiaolin Wu's line algorithm [Xiaolin, 1991]. This becomes especially obvious for 45 (or near 45) degree lines. Xiaolin Wu uses the fraction of the increment as intensity information of the pixel. For 45 degree lines no fraction occurs so no anti-aliasing is used. The lines appear therefore a bit thinner than the algorithm of listing 16.

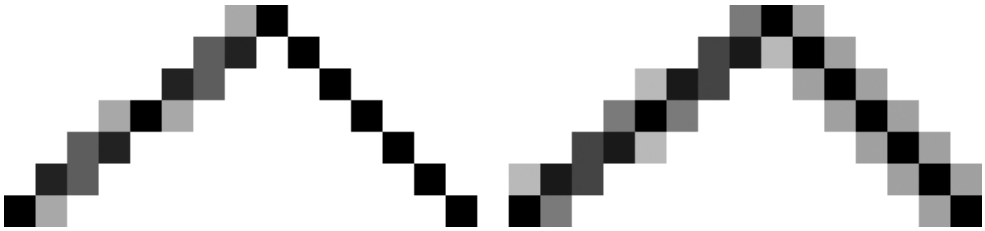


Figure 26: left Xiaolin Wu's, right this line algorithm

Though 45 degree lines are always a compromise on raster displays Xiaolin Wu's lines look sharper and the new algorithm a bit softer. It depends very much on the device and application.

It is also possible to make the lines of listing 16 to look a bit sharper by making the maximum error distance smaller. The transmission of the intensity values from one pixel to the next is then faster and uses less 'gray' values.

Another option would be to change the algorithm to accept floating point values as coordinate values. The line looks then like starting and ending at positions between the integer pixel.

7.2 Anti-aliased circle

The program for the anti-aliased circle is roughly the same than the first circle program in listing 5 except than each pixel becomes a gray value. In addition the x-step sets the outward pixel and the y-step sets the inward pixel of the circle.

```
void plotCircleAA(int xm, int ym, int r)
{             /* draw a black anti-aliased circle on white background */
   int x = -r, y = 0;   /* II. quadrant from bottom left to top right */
   int i, x2, e2, err = 2-2*r;                     /* error of 1.step */
   r = 1-err;
   do {
      i = 255*abs(err-2*(x+y)-2)/r;       /* get blend value of pixel */
      setPixelAA(xm-x, ym+y, i);                     /*   I. Quadrant */
      setPixelAA(xm-y, ym-x, i);                     /*  II. Quadrant */
      setPixelAA(xm+x, ym-y, i);                     /* III. Quadrant */
      setPixelAA(xm+y, ym+x, i);                     /*  IV. Quadrant */
      e2 = err; x2 = x;                             /* remember values */
      if (err+y > 0) {                                        /* x step */
         i = 255*(err-2*x-1)/r;                      /* outward pixel */
         if (i < 256) {
            setPixelAA(xm-x, ym+y+1, i);
            setPixelAA(xm-y-1, ym-x, i);
            setPixelAA(xm+x, ym-y-1, i);
            setPixelAA(xm+y+1, ym+x, i);
         }
         err += ++x*2+1;
      }
      if (e2+x2 <= 0) {                                       /* y step */
         i = 255*(2*y+3-e2)/r;                        /* inward pixel */
         if (i < 256) {
            setPixelAA(xm-x2-1, ym+y, i);
            setPixelAA(xm-y, ym-x2-1, i);
            setPixelAA(xm+x2+1, ym-y, i);
            setPixelAA(xm+y, ym+x2+1, i);
         }
         err += ++y*2+1;
      }
   } while (x < 0);
}
```

Listing 17: Program to plot an anti-aliased circle

This time the algorithm has no problem with spurious pixel.

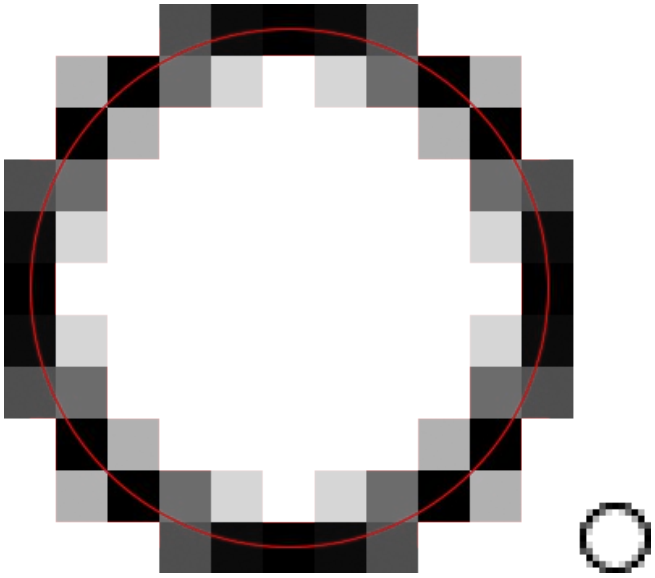


Figure 27: Example of an anti-aliased circle

7.3 Anti-aliased ellipse

The algorithm of the anti-aliased line has the benefit that the calculation of maximum error had only to be done once outside the pixel loop. For all other curves this value changes with every pixel and had to be calculated anew every time. The square root calculation of equation (45) inside a pixel loop is too time expensive.

For this reason an approximation for used for the maximum error calculation:

$$e_d = d_x + \frac{2\,d_x d_y^2}{4\,d_x^2 + d_y^2} \qquad \text{for } [d_y \le d_x] \tag{46}$$

The error of this approximation is less than 2% (exactly $\sqrt{2}-1.4$) and accurate enough for the pixel intensity.

Again the algorithm does not work at the end of the ellipse. The error calculation is one pixel ahead where the curve changes already direction. This case also would set a pixel beyond the y-direction due to the anti-aliasing. This would not matter since the intensity is too low to be visible. But due to the approximation of the gradient by equation (46) this solution does not work properly for small ellipses. The break condition is therefore inserted before the anti-aliased pixel are set and the ellipse finished by an additional loop.

The increment of the position also sets the anti-aliased pixel. The values of the one direction are therefore still needed unmodified when the other is updated. The increment is therefore divided into two parts. The first sets the pixel and the second updates the error values for the pixel calculation. An other (may be faster) option would be to use temporary values.

```
void plotEllipseRectAA(int x0, int y0, int x1, int y1)
{/* draw a black anti-aliased rectangular ellipse on white background */
   long a = abs(x1-x0), b = abs(y1-y0), b1 = b&1;         /* diameter */
   float dx = 4*(1-a)*b*b, dy = 4*(b1+1)*a*a;      /* error increment */
   float ed, i, err = dx+dy+b1*a*a;                /* error of 1.step */
   bool f;

   if (a == 0 || b == 0) return plotLine(x0,y0, x1,y1);
   if (x0 > x1) { x0 = x1; x1 += a; }/* if called with swapped points */
   if (y0 > y1) y0 = y1;                           /* .. exchange them */
   y0 += (b+1)/2; y1 = y0-b1;                       /* starting pixel */
   a = 8*a*a; b1 = 8*b*b;

   for (;;) {                     /* approximate ed=sqrt(dx*dx+dy*dy) */
      i = fmin(-dx,dy); ed = fmax(-dx,dy);
      ed = 255/(ed+2*ed*i*i/(4*ed*ed+i*i));
      i = ed*fabs(err-dx-dy);   /* get intensity value by pixel error */
      setPixelAA(x0,y0, i); setPixelAA(x0,y1, i);
      setPixelAA(x1,y0, i); setPixelAA(x1,y1, i);

      if (f = 2*err+dy >= 0) {          /* x step, remember condition */
         if (x0 >= x1) break;
         i = ed*(err-dx);
         if (i < 256) {
            setPixelAA(x0,y0+1, i); setPixelAA(x0,y1-1, i);
            setPixelAA(x1,y0+1, i); setPixelAA(x1,y1-1, i);
         }  /* do error increment later since values are still needed */
      }
      if (2*err+dx <= 0) {                                  /* y step */
         i = ed*(dy-err);
         if (i < 256) {
            setPixelAA(x0+1,y0, i); setPixelAA(x1-1,y0, i);
            setPixelAA(x0+1,y1, i); setPixelAA(x1-1,y1, i);
         }
         y0++; y1--; err += dy += a;
      }
      if (f) { x0++; x1--; err += dx += b1; }    /* x error increment */
   }
   if (--x0 == x1++)               /* too early stop of flat ellipses */
      while (y0-y1 <= b) {
         i = 255*fabs(err-dx-dy)/dx;      /* -> finish tip of ellipse */
         setPixelAA(x0,y0, i); setPixelAA(x1,y0++, i);
         setPixelAA(x0,y1, i); setPixelAA(x1,y1--, i);
         err += dy += a;
      }
}
```

Figure 18: Program to plot an anti-aliased rectangular ellipse

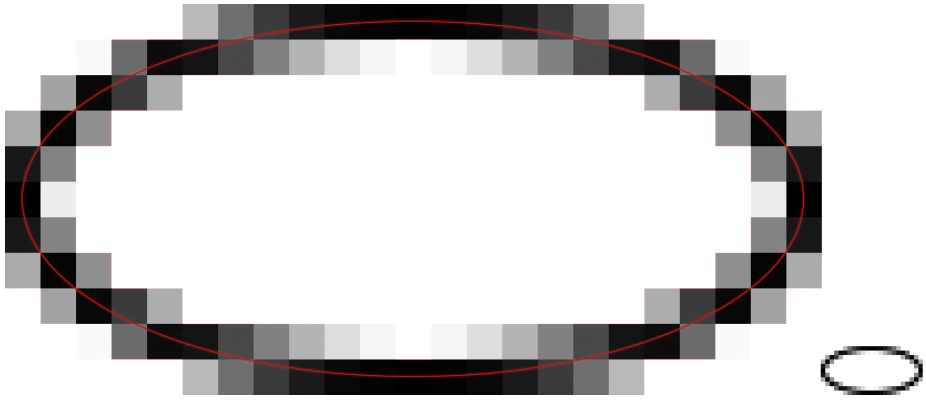


Figure 28: Example of an anti-aliased ellipse

A rotated ellipse could be drawn by using the anti-aliased rational quadratic Bézier curve of chapter 7.6.

7.4 Anti-aliased quadratic Bézier curve

The main procedure for drawing the anti-aliased quadratic Bézier curve remains the same as in chapter 3.7 listing 10. Only the subroutine for the Bézier segment changes slightly to set the pixel intensity of the curve according to the error value.

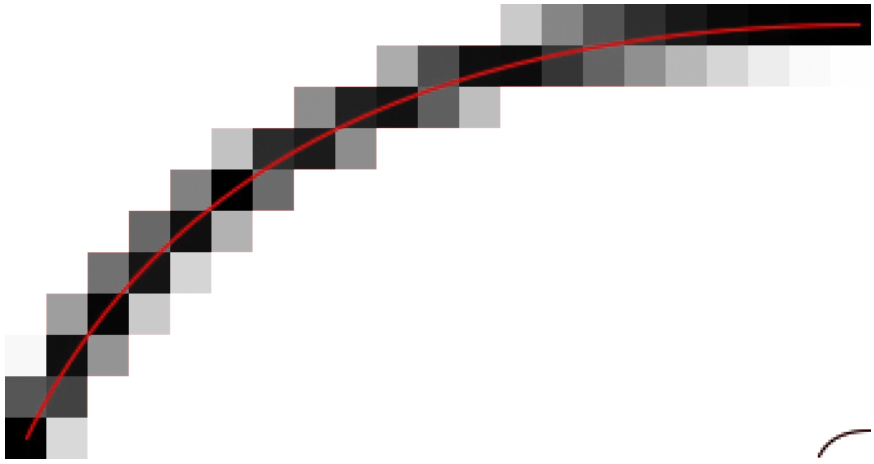


Figure 29: Example of an anti-aliased quadratic Bézier curve

The calculation of the maximum error distance also depends on the position. It makes a difference if the values of the current or next pixel are used. An exact calculation needs additionally the derivative of the curve. Using the difference values is an concession to get a faster and simpler algorithm.

Another slight error is introduced if the algorithm fails due to the other part of the curve comes close. In such case the curve is finished by an anti-aliased line but the transition from the curve to the line could not be exact. In addition the anti-aliased pixel values of the line to not exactly correspond to the values resulting from the Bézier curve calculation.

```
void plotQuadBezierSegAA(int x0, int y0, int x1, int y1, int x2, int y2)
{                /* draw an limited anti-aliased quadratic Bezier segment */
   int sx = x2-x1, sy = y2-y1;
   long xx = x0-x1, yy = y0-y1, xy;         /* relative values for checks */
   double dx, dy, err, ed, cur = xx*sy-yy*sx;                /* curvature */

   assert(xx*sx <= 0 && yy*sy <= 0);  /* sign of gradient must not change */

   if (sx*(long)sx+sy*(long)sy > xx*xx+yy*yy) { /* begin with longer part */
      x2 = x0; x0 = sx+x1; y2 = y0; y0 = sy+y1; cur = -cur; /* swap P0 P2 */
   }
   if (cur != 0)
   {                                                  /* no straight line */
      xx += sx; xx *= sx = x0 < x2 ? 1 : -1;          /* x step direction */
      yy += sy; yy *= sy = y0 < y2 ? 1 : -1;          /* y step direction */
      xy = 2*xx*yy; xx *= xx; yy *= yy;         /* differences 2nd degree */
      if (cur*sx*sy < 0) {                           /* negated curvature? */
         xx = -xx; yy = -yy; xy = -xy; cur = -cur;
      }
      dx = 4.0*sy*(x1-x0)*cur+xx-xy;            /* differences 1st degree */
      dy = 4.0*sx*(y0-y1)*cur+yy-xy;
      xx += xx; yy += yy; err = dx+dy+xy;               /* error 1st step */
      do {
         cur = fmin(dx+xy,-xy-dy);
         ed = fmax(dx+xy,-xy-dy);           /* approximate error distance */
         ed += 2*ed*cur*cur/(4*ed*ed+cur*cur);
         setPixelAA(x0,y0, 255*fabs(err-dx-dy-xy)/ed);      /* plot curve */
         if (x0 == x2 || y0 == y2) break;               /* curve finished */
         x1 = x0; cur = dx-err; y1 = 2*err+dy < 0;
         if (2*err+dx > 0) {                                    /* x step */
            if (err-dy < ed) setPixelAA(x0,y0+sy, 255*fabs(err-dy)/ed);
            x0 += sx; dx -= xy; err += dy += yy;
         }
         if (y1) {                                              /* y step */
            if (cur < ed) setPixelAA(x1+sx,y0, 255*fabs(cur)/ed);
            y0 += sy; dy -= xy; err += dx += xx;
         }
      } while (dy < dx);              /* gradient negates -> close curves */
   }
   plotLineAA(x0,y0, x2,y2);              /* plot remaining needle to end */
}
```

Listing 19: Program to plot an anti-aliased quadratic Bézier curve

It would also be possible to change the algorithm to accept curves with floating point arguments as coordinates for this subroutine. This would result in smoother curves.

7.5 Anti-aliased rational quadratic Bézier curve

The algorithm of the anti-aliased rational quadratic Bézier curve remains mainly the same than for the aliased or non rational curve algorithm since it faces the same problems. The curve must be divided if the weight is too small and the other part of the curve comes too close. Also the curve must be finished by the line algorithm for the tips the quadratic Bézier curves. The anti-aliasing is handled by setting pixel according the relative error value if the x or y direction is incremented.

Using the error value as measurement of the pixel distance to the curve is an approximation. As previously demonstrated the drawing algorithm fails if another part of the curve comes too close. This other part of the curve also influences the error values used for anti-aliasing significantly. So before the algorithm starts to fail the anti-aliasing begins to look incorrect some times.

A possible solution to this problem is to use the high resolution raster of chapter 3.4 to determine to increment direction.

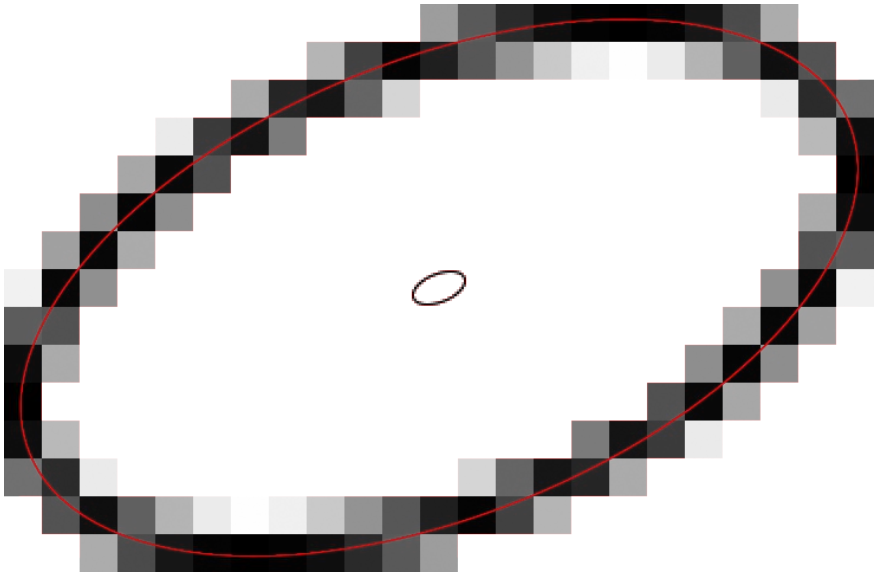


Figure 30: Example of an anti-aliased rotated ellipse

Figure 30 shows an anti-aliased rotated ellipse by a composition of four anti-aliased rational Bézier curves. The ellipse is divided by the program of listing 13 into rational quadratic Bézier segments which are drawn by the following program for rational anti-aliased quadratic Bézier segments.

The calculation of the maximum error distance again uses the approximation of equation 46 to avoid the expensive square root function in the pixel loop.

```
void plotQuadRationalBezierSegAA(int x0, int y0, int x1, int y1,
                                 int x2, int y2, float w)
{    /* draw an anti-aliased rational quadratic Bezier segment, squared weight */
   int sx = x2-x1, sy = y2-y1;                    /* relative values for checks */
   double dx = x0-x2, dy = y0-y2, xx = x0-x1, yy = y0-y1;
   double xy = xx*sy+yy*sx, cur = xx*sy-yy*sx, err, ed;           /* curvature */
   bool f;
   assert(xx*sx <= 0.0 && yy*sy <= 0.0);   /* sign of gradient must not change */

   if (cur != 0.0 && w > 0.0) {                              /* no straight line */
      if (sx*(long)sx+sy*(long)sy > xx*xx+yy*yy) {   /* begin with longer part */
         x2 = x0; x0 -= dx; y2 = y0; y0 -= dy; cur = -cur;       /* swap P0 P2 */
      }
      xx = 2.0*(4.0*w*sx*xx+dx*dx);                    /* differences 2nd degree */
      yy = 2.0*(4.0*w*sy*yy+dy*dy);
      sx = x0 < x2 ? 1 : -1;                                /* x step direction */
      sy = y0 < y2 ? 1 : -1;                                /* y step direction */
      xy = -2.0*sx*sy*(2.0*w*xy+dx*dy);

      if (cur*sx*sy < 0) {                                /* negated curvature? */
         xx = -xx; yy = -yy; cur = -cur; xy = -xy;
      }
      dx = 4.0*w*(x1-x0)*sy*cur+xx/2.0+xy;             /* differences 1st degree */
      dy = 4.0*w*(y0-y1)*sx*cur+yy/2.0+xy;

      if (w < 0.5 && dy > dx) {                 /* flat ellipse, algorithm fails */
         cur = (w+1.0)/2.0; w = sqrt(w); xy = 1.0/(w+1.0);
         sx = floor((x0+2.0*w*x1+x2)*xy/2.0+0.5);  /* subdivide curve in half  */
         sy = floor((y0+2.0*w*y1+y2)*xy/2.0+0.5);
         dx = floor((w*x1+x0)*xy+0.5); dy = floor((y1*w+y0)*xy+0.5);
         plotQuadRationalBezierSegAA(x0,y0, dx,dy, sx,sy, cur);  /* plot apart */
         dx = floor((w*x1+x2)*xy+0.5); dy = floor((y1*w+y2)*xy+0.5);
         return plotQuadRationalBezierSegAA(sx,sy, dx,dy, x2,y2, cur);
      }
      err = dx+dy-xy;                                        /* error 1st step */
      do {                                                       /* pixel loop */
         cur = fmin(dx-xy,xy-dy); ed = fmax(dx-xy,xy-dy);
         ed += 2*ed*cur*cur/(4.*ed*ed+cur*cur);  /* approximate error distance */
         x1 = 255*fabs(err-dx-dy+xy)/ed;     /* get blend value by pixel error */
         if (x1 < 256) setPixelAA(x0,y0, x1);                    /* plot curve */
         if (f = 2*err+dy < 0) {                                     /* y step */
            if (y0 == y2) return;              /* last pixel -> curve finished */
            if (dx-err < ed) setPixelAA(x0+sx,y0, 255*fabs(dx-err)/ed);
         }
         if (2*err+dx > 0) {                                         /* x step */
            if (x0 == x2) return;              /* last pixel -> curve finished */
            if (err-dy < ed) setPixelAA(x0,y0+sy, 255*fabs(err-dy)/ed);
            x0 += sx; dx += xy; err += dy += yy;
         }
         if (f) { y0 += sy; dy += xy; err += dx += xx; }            /* y step */
      } while (dy < dx);                /* gradient negates -> algorithm fails */
   }
   plotLineAA(x0,y0, x2,y2);                   /* plot remaining needle to end */
}
```

Listing 20: Program to plot an anti-aliased rational quadratic Bézier curve

7.6 Anti-aliased cubic Bézier curve

The algorithm of the anti-aliased cubic Bézier curve runs into more troubles than the normal cubic one. The reason is the same than previously: the points on the 3D surface where the derivation gets zero or infinity. At this points it is not possible to calculate the one pixel distance of the difference error values.

In the aliased version of the drawing algorithm it was sufficient to keep the drawing direction on track of the curve. Now this is not enough. The intensity information of the pixel is needed additionally.

The algorithm of listing 14 detects a near crossing of the curve by a change of the difference values. Since the increment condition of the anti-aliased algorithm is different the detection of the near crossing also changes. It only checks for a change of the difference values two pixel ahead if the curve contains a self-intersection loop at all to avoid a break in case of difficult start conditions.

Another difficult problem is the high resolution raster which is used inside the self-intersection loop. The algorithm needs the error values of pixel as intensity information but the calculation runs along the finer resolution. The error value must therefore be calculated depending on the current position of the sub pixel, not to forget that these values also change with every sub step.

Figure 31: Example of an anti-aliased cubic Bézier curve

Figure 31 shows an anti-aliased cubic Bézier curve. The anti-aliasing seems to be a bit strange at the top left corner. The reason is that the sharp edge in the curve comes from an almost singular point.

Basically the algorithm assumes that the differences do not change near the set pixel. This is correct in most cases. But on singular or near singular points like cusps this is not true. The difference values change very much in these areas. Especially the maximum error

value as reference for the pixel distance changes too fast with every pixel in such cases. The intensity value for the pixel nearby is therefore not in any case the best solution for this simple algorithm.

```
void plotCubicBezierSegAA(int x0, int y0, float x1, float y1,
                          float x2, float y2, int x3, int y3)
{                         /* plot limited anti-aliased cubic Bezier segment */
   int f, fx, fy, leg = 1;
   int sx = x0 < x3 ? 1 : -1, sy = y0 < y3 ? 1 : -1;      /* step direction */
   float xc = -fabs(x0+x1-x2-x3), xa = xc-4*sx*(x1-x2), xb = sx*(x0-x1-x2+x3);
   float yc = -fabs(y0+y1-y2-y3), ya = yc-4*sy*(y1-y2), yb = sy*(y0-y1-y2+y3);
   double ab, ac, bc, ba, xx, xy, yy, dx, dy, ex, px, py, ed, ip, EP = 0.01;

                                              /* check for curve restrains */
   /* slope P0-P1 == P2-P3   and  (P0-P3 == P1-P2      or  no slope change) */
   assert((x1-x0)*(x2-x3) < EP && ((x3-x0)*(x1-x2) < EP || xb*xb < xa*xc+EP));
   assert((y1-y0)*(y2-y3) < EP && ((y3-y0)*(y1-y2) < EP || yb*yb < ya*yc+EP));

   if (xa == 0 && ya == 0) {                            /* quadratic Bezier */
      sx = floor((3*x1-x0+1)/2); sy = floor((3*y1-y0+1)/2); /* new midpoint */
      return plotQuadBezierSegAA(x0,y0, sx,sy, x3,y3);
   }
   x1 = (x1-x0)*(x1-x0)+(y1-y0)*(y1-y0)+1;                 /* line lengths */
   x2 = (x2-x3)*(x2-x3)+(y2-y3)*(y2-y3)+1;
   do {                                                /* loop over both ends */
      ab = xa*yb-xb*ya; ac = xa*yc-xc*ya; bc = xb*yc-xc*yb;
      ip = 4*ab*bc-ac*ac;                 /* self intersection loop at all? */
      ex = ab*(ab+ac-3*bc)+ac*ac;     /* P0 part of self-intersection loop? */
      f = ex > 0 ? 1 : sqrt(1+1024/x1);             /* calculate resolution */
      ab *= f; ac *= f; bc *= f; ex *= f*f;          /* increase resolution */
      xy = 9*(ab+ac+bc)/8; ba = 8*(xa-ya);/* init differences of 1st degree */
      dx = 27*(8*ab*(yb*yb-ya*yc)+ex*(ya+2*yb+yc))/64-ya*ya*(xy-ya);
      dy = 27*(8*ab*(xb*xb-xa*xc)-ex*(xa+2*xb+xc))/64-xa*xa*(xy+xa);
                                          /* init differences of 2nd degree */
      xx = 3*(3*ab*(3*yb*yb-ya*ya-2*ya*yc)-ya*(3*ac*(ya+yb)+ya*ba))/4;
      yy = 3*(3*ab*(3*xb*xb-xa*xa-2*xa*xc)-xa*(3*ac*(xa+xb)+xa*ba))/4;
      xy = xa*ya*(6*ab+6*ac-3*bc+ba); ac = ya*ya; ba = xa*xa;
      xy = 3*(xy+9*f*(ba*yb*yc-xb*xc*ac)-18*xb*yb*ab)/8;

      if (ex < 0) {       /* negate values if inside self-intersection loop */
         dx = -dx; dy = -dy; xx = -xx; yy = -yy; xy = -xy; ac = -ac; ba = -ba;
      }                                   /* init differences of 3rd degree */
      ab = 6*ya*ac; ac = -6*xa*ac; bc = 6*ya*ba; ba = -6*xa*ba;
      dx += xy; ex = dx+dy; dy += xy;                  /* error of 1st step */

      for (fx = fy = f; x0 != x3 && y0 != y3; ) {
         y1 = fmin(xy-dx,dy-xy);
         ed = fmax(xy-dx,dy-xy);              /* approximate error distance */
         ed = f*(ed+2*ed*y1*y1/(4*ed*ed+y1*y1));
```

```
45          y1 = 255*fabs(ex-(f-fx+1)*dx-(f-fy+1)*dy+f*xy)/ed;
            if (y1 < 256) setPixelAA(x0, y0, y1);               /* plot curve */
            px = fabs(ex-(f-fx+1)*dx+(fy-1)*dy);     /* pixel intensity x move */
            py = fabs(ex+(fx-1)*dx-(f-fy+1)*dy);     /* pixel intensity y move */
            y2 = y0;
50          do {                                  /* move sub-steps of one pixel */
               if (ip >= -EP)             /* intersection possible? -> check.. */
                  if (dx+xx > xy || dy+yy < xy) goto exit; /* two x or y steps */
               y1 = 2*ex+dx;                   /* save value for test of y step */
               if (2*ex+dy > 0) {                                  /* x sub-step */
55                fx--; ex += dx += xx; dy += xy += ac; yy += bc; xx += ab;
               } else if (y1 > 0) goto exit;               /* tiny nearly cusp */
               if (y1 <= 0) {                                      /* y sub-step */
                  fy--; ex += dy += yy; dx += xy += bc; xx += ac; yy += ba;
               }
60          } while (fx > 0 && fy > 0);                     /* pixel complete? */
            if (2*fy <= f) {                          /* x+ anti-aliasing pixel */
               if (py < ed) setPixelAA(x0+sx, y0, 255*py/ed);    /* plot curve */
               y0 += sy; fy += f;                                   /* y step */
            }
65          if (2*fx <= f) {                          /* y+ anti-aliasing pixel */
               if (px < ed) setPixelAA(x0, y2+sy, 255*px/ed);    /* plot curve */
               x0 += sx; fx += f;                                   /* x step */
            }
         }
70       break;                                       /* finish curve by line */
exit:
         if (2*ex < dy && 2*fy <= f+2) {       /* round x+ approximation pixel */
            if (py < ed) setPixelAA(x0+sx, y0, 255*py/ed);       /* plot curve */
            y0 += sy;
75       }
         if (2*ex > dx && 2*fx <= f+2) {       /* round y+ approximation pixel */
            if (px < ed) setPixelAA(x0, y2+sy, 255*px/ed);       /* plot curve */
            x0 += sx;
         }
80       xx = x0; x0 = x3; x3 = xx; sx = -sx; xb = -xb;           /* swap legs */
         yy = y0; y0 = y3; y3 = yy; sy = -sy; yb = -yb; x1 = x2;
      } while (leg--);                                        /* try other end */
      plotLineAA(x0,y0, x3,y3);   /* remaining part in case of cusp or crunode */
}
```

Listing 21: Program to plot an anti-aliased cubic Bézier curve

The algorithm in listing 21 is not perfect. On curve intersections the transition from the curve to the line approximation remains slightly visible.

In case of a self-intersection the algorithm may be in the middle of the anti-aliasing. Now the calculation of the step direction fails and the drawing continues with an anti-aliasing line. This also breaks the intensity calculation of the pixel.

Figure 32: Troublesome anti-aliasing cubic Bézier curves

Figure 32 shows a few problematic samples of cubic Bézier curves. If you look carefully you will some-times notice some strange anti-aliasing pixel at the curve. This is always near a self-intersection point where the strategy of the algorithm must change from cubic curve to straight line.

To overcome these deficiencies the self-intersection loop needs more advanced (and complex) algorithm.

8 Thick anti-aliased curve

Drawing a thick anti-aliased curve could be done in different ways. One possibility is to first set the pixel of full intensity and after wards add the pixel of the anti-aliased intensity. Or to draw the left and right side of the curve and fill the space between them.

The algorithm of this document could also be used to draw thick curves. The information of the error value is used to calculate the distance from the curve and set the intensity of the pixels according this value.

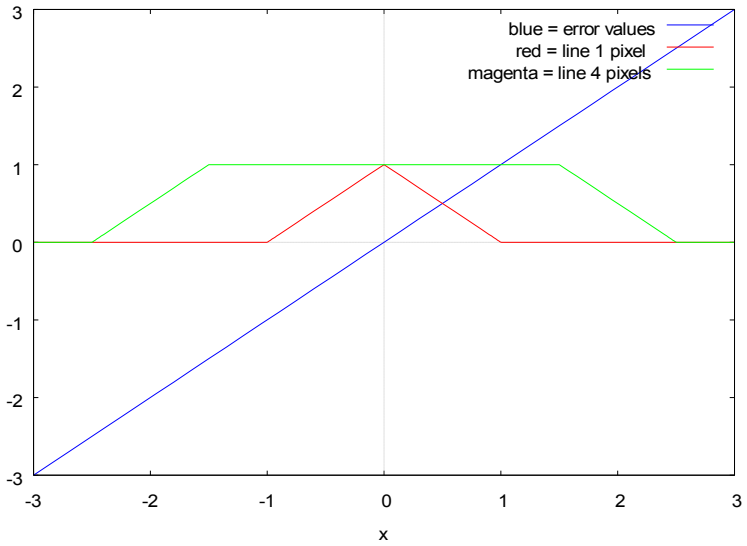


Figure 33: Line thickness according to error values

Figure 33 show how the error value is used to determine the intensity value of the pixel to draw lines of different widths. The horizontal axis refers the relative error value and the vertical axis the pixel intensity.

8.1 Thick lines

The simplest realization of the above statement are lines of certain thickness. A fast implementation would be to draw the left and right line simultaneously and fill in the pixels between.

The next listing 22 demonstrates the method by using the error value as a distance measurement. In addition to the anti-aliasing version not only one pixel has to be set whenever the step increases but several points.

In case of a straight horizontal or vertical line the algorithm must be careful not to draw beyond of the end pixel when thickness of line sets the pixels.

```
void plotLineWidth(int x0, int y0, int x1, int y1, float wd)
{                                    /* plot an anti-aliased line of width wd */
   int dx = abs(x1-x0), sx = x0 < x1 ? 1 : -1;
   int dy = abs(y1-y0), sy = y0 < y1 ? 1 : -1;
   int err = dx-dy, e2, x2, y2;                          /* error value e_xy */
   float ed = dx+dy == 0 ? 1 : sqrt((float)dx*dx+(float)dy*dy);

   for (wd = (wd+1)/2; ; ) {                                    /* pixel loop */
      setPixelColor(x0, y0, max(0,255*(abs(err-dx+dy)/ed-wd+1)));
      e2 = err; x2 = x0;
      if (2*e2 >= -dx) {                                           /* x step */
         for (e2 += dy, y2 = y0; e2 < ed*wd && (y1 != y2 || dx > dy); e2 += dx)
            setPixelColor(x0, y2 += sy, max(0,255*(abs(e2)/ed-wd+1)));
         if (x0 == x1) break;
         e2 = err; err -= dy; x0 += sx;
      }
      if (2*e2 <= dy) {                                            /* y step */
         for (e2 = dx-e2; e2 < ed*wd && (x1 != x2 || dx < dy); e2 += dy)
            setPixelColor(x2 += sx, y0, max(0,255*(abs(e2)/ed-wd+1)));
         if (y0 == y1) break;
         err += dx; y0 += sy;
      }
   }
}
```

Listing 22: Program to plot an anti-aliased thick line

Curves of thick lines need special attention at the end points. The type of the end points mostly depend on the application. Different possibilities are rectangle, round, arrows or special customized. This is also important if the line is continued either by another line or or another type of curve like arcs, or Bézier curves of higher degree. It is therefore not possible to present an universal algorithm for all types. None the less an adaption to different requirements should be possible.

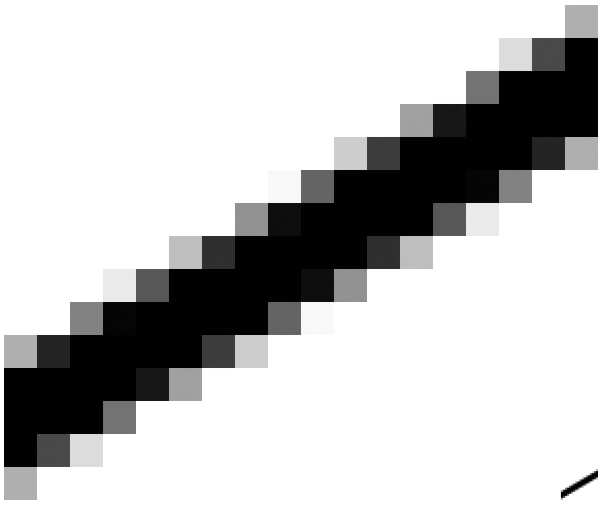


Figure 34: Thick line with anti-aliasing

8.2 Thick curves of higher degree

The previous algorithm could also be applied to curves of higher degree. This works straight forward as long as no other curve comes close. Since this type of algorithm used the error values farther away from the curve itself the assumption that the difference values do not change will fail more likely. It gets therefore very difficult to draw more complex curves with sharp edges, intersections and loops. Whenever another part of the curve comes too close the differences change too much and more advanced techniques must be used to compensate the flaw.

For simple figures like circles and ellipses it is easier to draw the inner and outer edge of the curve and fill in the space between.

9 Splines

Single Bézier curves of higher degree than cubic get very complicated to draw. More complex curves are made by connections of quadratic or cubic Bézier curves.

Two adjacent curves are C^k continuous if the 0^{th} to k-th derivative of the curve is equal at the connection point. Thus, C^0 continuity simply means the two adjacent curves share a common point. C^1 continuity means that the two curves have in addition the same tangent vector. C^2 continuity means that the second order parametric derivatives of the shared endpoint equals in magnitude and in direction.

Splines offer a great flexibility by a minimum of control handles making editing of curves easier. Simpler curves with fewer control points are B-splines. Only the corner points of the curve are defined. The additional control points are placed to automatically join the curves continuously. [Piegl et al., 1996]

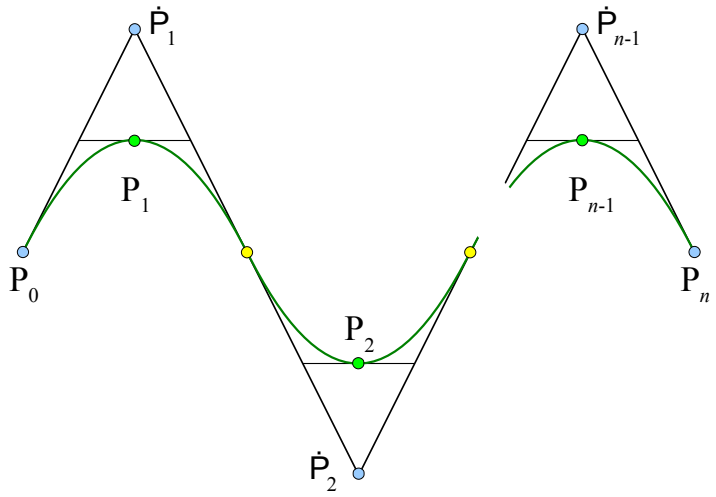


Figure 35: Quadratic spline

Splines of degree one are a polygon line.

9.1 Quadratic B-splines

If the corner points $\dot{P}_i$ in figure 35 are given the drawing of the curve is straight forward. (The yellow points are half way between the cyan.) Is it possible to calculate the points $\dot{P}_i$ (cyan) if the points P_i (green) *at* the curve are given?

In case the green Points at the curve are given the corner points must be computed by:

$$\dot{P}_{i-1}+6\dot{P}_i+\dot{P}_{i+1}=8P_i,\quad 5\dot{P}_1+\dot{P}_2=8P_1-2P_0,\quad \dot{P}_{n-2}+5\dot{P}_{n-1}=8P_{n-1}-2P_n.$$

The equations can be written in matrix form:

$$\begin{bmatrix} 5 & 1 & & & 0 \\ 1 & 6 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 6 & 1 \\ 0 & & & 1 & 5 \end{bmatrix} \cdot \begin{bmatrix} \dot{\mathrm{P}}_1 \\ \dot{\mathrm{P}}_2 \\ \vdots \\ \dot{\mathrm{P}}_{n-2} \\ \dot{\mathrm{P}}_{n-1} \end{bmatrix} = \begin{bmatrix} 8\,\mathrm{P}_1 - 2\,\mathrm{P}_0 \\ 8\,\mathrm{P}_2 \\ \vdots \\ 8\,\mathrm{P}_{n-2} \\ 8\,\mathrm{P}_{n-1} - 2\,\mathrm{P}_n \end{bmatrix} \tag{47}$$

This tridiagonal systems of linear equations could be solved by the Thomas algorithm in $\mathcal{O}(n)$ operations by LU decomposition of the matrix $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{d}$ [Thomas, 1949]:

$$\begin{bmatrix} a_1 & b_1 & & & 0 \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & & & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & & & & 0 \\ l_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{n-1} & 1 & \\ 0 & & & l_n & 1 \end{bmatrix} \cdot \begin{bmatrix} m_1 & b_1 & & & 0 \\ & m_2 & b_2 & & \\ & & \ddots & \ddots & \\ & & & m_{n-1} & b_{n-1} \\ 0 & & & & m_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{48}$$

The matrix equation is now solved by the steps: $\boldsymbol{L} \cdot \boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{d}$, $\boldsymbol{L} \cdot \boldsymbol{y} = \boldsymbol{d}$ and $\boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{y}$.

Comparing the terms the unknowns could be calculated by a forward sweep:
$m_1 = a_1$, $l_{i+1} = c_{i+1}/m_i$, $m_{i+1} = a_{i+1} - b_i\, l_{i+1}$ and the temporary result $y_1 = d_1$, $y_i = d_i - y_{i-1}\, l_i$.

The back substitution finally calculates the desired result: $x_n = y_n/m_n$, $x_{i-1} = (y_{i-1} - b_{i-1}\, x_i)/m_{i-1}$.

If the input arrays could be used as temporary buffer no extra storage is needed by the algorithm. If this is not desirable new arrays must be allocated to store the intermediate results.

```
void plotQuadSpline(int n, int x[], int y[])
{            /* plot quadratic spline, destroys input arrays x,y */
   #define M_MAX 6
   float mi = 1, m[M_MAX];       /* diagonal constants of matrix */
   int i, x0, y0, x1, y1, x2 = x[n], y2 = y[n];

   assert(n > 1);           /* need at least 3 points P[0]..P[n] */

   x[1] = x0 = 8*x[1]-2*x[0];              /* first row of matrix */
   y[1] = y0 = 8*y[1]-2*y[0];

   for (i = 2; i < n; i++) {                     /* forward sweep */
      if (i-2 < M_MAX) m[i-2] = mi = 1.0/(6.0-mi);
      x[i] = x0 = floor(8*x[i]-x0*mi+0.5);           /* store yi */
      y[i] = y0 = floor(8*y[i]-y0*mi+0.5);
   }
   x1 = floor((x0-2*x2)/(5.0-mi)+0.5);     /* correction last row */
   y1 = floor((y0-2*y2)/(5.0-mi)+0.5);

   for (i = n-2; i > 0; i--) {               /* back substitution */
      if (i <= M_MAX) mi = m[i-1];
      x0 = floor((x[i]-x1)*mi+0.5);                /* next corner */
      y0 = floor((y[i]-y1)*mi+0.5);
      plotQuadBezier((x0+x1)/2,(y0+y1)/2, x1,y1, x2,y2);
      x2 = (x0+x1)/2; x1 = x0;
      y2 = (y0+y1)/2; y1 = y0;
   }
   plotQuadBezier(x[0],y[0], x1,y1, x2,y2);
}
```

Listing 23: Drawing a quadratic spline given points P_i at curve

This program even works if only a single Bézier curve has to be drawn ($n=2$).

The algorithm needs a temporary floating point array to hold the diagonal values of the matrix: $m_1=5,\ m_{i+1}=6-1/m_i$. But since the sequence soon limits to the constant of $\lim_{i\to\infty} m_i=3+\sqrt{8}$ the first 6 entries are accurate enough.

The Thomas algorithm could also be used for periodic splines.

9.2 Cubic splines

Figure 36 shows a cubic spline with two different end conditions. The control points are labeled in polar notation. Point $P_{0,0,0}$ is a double point where the curvature equals zero.

A few possibilities exist for the end condition of the curve:

- open end: the curve is just drawn from $P_{1,1,1}$ to $P_{n\text{-}1,n\text{-}1,n\text{-}1}$.
- tangent: the curvature at the end points $P_{0,0,0}$ and $P_{n,n,n}$ vanishes.
- periodic: the curve is closed by $P_{0,0,0} = P_{n,n,n}$.

In case the corner points (cyan) are given the curve could be subdivided in cubic Bézier curves using DeBoor's algorithm. A cubic Bézier curve is extracted by the Points $P_{i,i,i}$-$P_{i,i,i+1}$-$P_{i,i+1,i+1}$-$P_{i+1,i+1,i+1}$. The line $P_{i\text{-}1,i,i+1}$-$P_{i,i+1,i+2}$ is made thirds to get $P_{i,i,i+1}$ and $P_{i,i+1,i+1}$. The line $P_{i\text{-}1,i,i}$-$P_{i,i,i+1}$ is halved to get $P_{i,i,i}$.

The equations for cubic splines are similar:

$$\dot{P}_{i-1}+4\dot{P}_i+\dot{P}_{i+1}=6P_i,\quad 7\dot{P}_1+2\dot{P}_2=12P_1-3P_0,\quad 2\dot{P}_{n-2}+7\dot{P}_{n-1}=12P_{n-1}-3P_n.$$

$$\begin{bmatrix} 7 & 2 & & & 0 \\ 2 & 8 & 2 & & \\ & \ddots & \ddots & \ddots & \\ & & 2 & 8 & 2 \\ 0 & & & 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} \dot{P}_1 \\ \dot{P}_2 \\ \vdots \\ \dot{P}_{n-2} \\ \dot{P}_{n-1} \end{bmatrix} = \begin{bmatrix} 12P_1 - 3P_0 \\ 12P_2 \\ \vdots \\ 12P_{n-2} \\ 12P_{n-1} - 3P_n \end{bmatrix} \tag{49}$$

In case the through points (yellow) are given it is again possible to compute the corner points. But this system of linear equations has two degrees of freedom. A possible condition for the needed restriction is to define the curvature at both ends to be zero (multiple knots): B"(0) = 0. This condition is shown at the beginning of the curve in figure 36 ($P_{0,0,0}$).

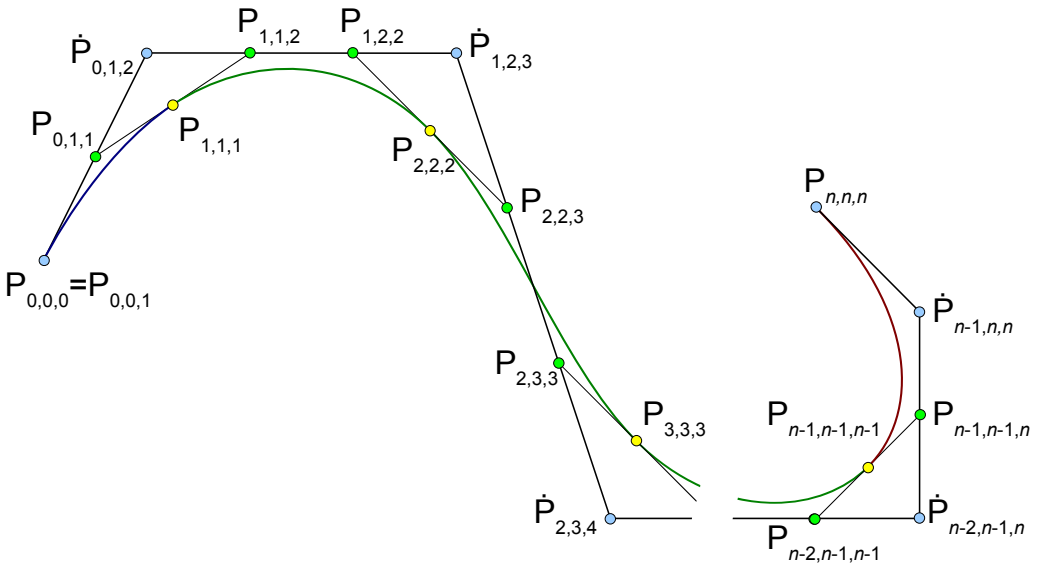


Figure 36: Cubic spline

Again the Thomas algorithm helps to solve the matrix equation.

```
void plotCubicSpline(int n, int x[], int y[])
{                       /* plot cubic spline, destroys input arrays x,y */
   #define M_MAX 6
   float mi = 0.25, m[M_MAX];          /* diagonal constants of matrix */
   int x3 = x[n-1], y3 = y[n-1], x4 = x[n], y4 = y[n];
   int i, x0, y0, x1, y1, x2, y2;

   assert(n > 2);                 /* need at least 4 points P[0]..P[n] */

   x[1] = x0 = 12*x[1]-3*x[0];                   /* first row of matrix */
   y[1] = y0 = 12*y[1]-3*y[0];

   for (i = 2; i < n; i++) {                          /* foreward sweep */
      if (i-2 < M_MAX) m[i-2] = mi = 0.25/(2.0-mi);
      x[i] = x0 = floor(12*x[i]-2*x0*mi+0.5);
      y[i] = y0 = floor(12*y[i]-2*y0*mi+0.5);
   }
   x2 = floor((x0-3*x4)/(7-4*mi)+0.5);             /* correct last row */
   y2 = floor((y0-3*y4)/(7-4*mi)+0.5);
   plotCubicBezier(x3,y3, (x2+x4)/2,(y2+y4)/2, x4,y4, x4,y4);

   if (n-3 < M_MAX) mi = m[n-3];
   x1 = floor((x[n-2]-2*x2)*mi+0.5);
   y1 = floor((y[n-2]-2*y2)*mi+0.5);
   for (i = n-3; i > 0; i--) {                    /* back substitution */
      if (i <= M_MAX) mi = m[i-1];
      x0 = floor((x[i]-2*x1)*mi+0.5);
      y0 = floor((y[i]-2*y1)*mi+0.5);
      x4 = floor((x0+4*x1+x2+3)/6.0);              /* reconstruct P[i] */
      y4 = floor((y0+4*y1+y2+3)/6.0);
      plotCubicBezier(x4,y4,
                      floor((2*x1+x2)/3+0.5),floor((2*y1+y2)/3+0.5),
                      floor((x1+2*x2)/3+0.5),floor((y1+2*y2)/3+0.5),
                      x3,y3);
      x3 = x4; y3 = y4; x2 = x1; y2 = y1; x1 = x0; y1 = y0;
   }
   x0 = x[0]; x4 = floor((3*x0+7*x1+2*x2+6)/12.0); /* reconstruct P[1] */
   y0 = y[0]; y4 = floor((3*y0+7*y1+2*y2+6)/12.0);
   plotCubicBezier(x4,y4, floor((2*x1+x2)/3+0.5),floor((2*y1+y2)/3+0.5),
                   floor((x1+2*x2)/3+0.5),floor((y1+2*y2)/3+0.5), x3,y3);
   plotCubicBezier(x0,y0, x0,y0, (x0+x1)/2,(y0+y1)/2, x4,y4);
}
```

Listing 24: Drawing a cubic spline given points at *curve*

The algorithm needs an additional floating point array to store the values of the forward sweep since the curve points are still needed to draw the computed Bézier segment. But the original points could be reconstructed by the next corner point avoiding the additional array.

Please note that the cubic spline of size $n = 3$ drawn with this algorithm is not the same than a single cubic Bézier although both are defined by 4 points. But the condition for this spline is in addition to have end points without a curvature, whereas the cubic Bézier does not have such a restriction.

10 Conclusions

Rasterizing is a fundamental task in computer graphics. This document reworks the Bresenham algorithm and extends it for other curves like ellipses and Bézier curves of quadratic and cubic degree. Generally the algorithm could be used to rasterize any geometric curve.

10.1 Algorithm to plot implicit equations

The calculations of the previous chapters help to define a common algorithm for the plotting curves of the implicit equation $\mathrm{f}(x,y)=0$.

1. Since the plotting algorithm relies on a continuously positive or negative gradient the curve must be subdivided at stationary points at which the gradient changes its sign (or possibly at self intersections).
   The change of the direction could also be included in the pixel loop. The subdivision is not necessary as long as the curve turn only occurs in one direction at the same time (either x or y). A change in the direction could be detected by a sign change of the differences. The increment values are adapted according the new direction. This solution only runs into troubles if both drawing directions change simultaneously since it cannot decide which of the curves to follow.
2. The step increment is computed by successive differences of the implicit equation: in x-direction: $d_{x,y}$ += $d_{x+1,y}$, err += d_{10} and in y-direction: $d_{x,y}$ += $d_{x,y+1}$, err += d_{01}.
3. These increments are initialized by the differences of the starting pixel. The differences of the error values (of spacing h) are the finite m-th order forward differences in two dimensions:
   $$d_{n_x n_y}=\frac{1}{h_x^{n_x} h_y^{n_y}}\sum_{i_x=0}^{n_x}\sum_{i_y=0}^{n_y}(-1)^{i_x+i_y}\binom{n_x}{i_x}\binom{n_y}{i_y}\mathrm{f}(x+(n_x-i_x)h_x,\, y+(n_y-i_y)h_y)\quad [n_x+n_y=m].$$
   The initialization value of the polynomial equation of degree *n* makes the *n*-th derivative of the implicit equation: $d_{n_x,n_y}=\dfrac{\partial^n \mathrm{f}}{\partial x^{n_x}\partial y^{n_y}}\quad [n_x+n_y=n].$
4. The algorithm must take special care if another part of the function comes close to the set pixel of the curve.

The algorithm could be used on all functions with implicit polynomial equation.

10.2 Algorithm complexity

Since any continuous function can be approximated by a polynomial function (Weierstrass theorem) this document provides an algorithm to draw any function nearly as fast as a line. More complex curves only need more additions per pixel.

An algorithm of degree one (line) only adds the difference of the error value for every step. The algorithm of degree two (quadratic) additionally has to track the changes of the differences d_x, d_y. (Ellipses and circles are simpler since certain differences are equal due to the symmetry of the curve.) The algorithm of the third degree also has to track the changes of the changes d_{xx}, d_{xy}, d_{yy}.

The number of needed operations drawing a polynomial curve of degree n makes in Big O notation: $n(n+1)/2 = \mathcal{O}(n^2)$.
The implementation of the algorithm is limited by the complex calculations of the initialization values for the differences. Especially for curves of higher degrees these expressions can get quite large.

10.3 Applications

Rasterizing is a fundamental task in computer graphics. Vector graphic is based on geometrical primitives such as lines, circles, ellipses and Bezier curves. Such curves must be rasterized on every output device like displays, printers, plotters, machines, etc.

The algorithm of this document makes the drawing of curves computationally efficient and it is also very simple to implement. An everyday program may delegate the drawing to a subroutine, the operating system or basic input/output driver. But if an application wants to have control over the drawing process or must have access to certain details of the curve parameter it needs to implement the plot algorithm itself.

This type of algorithm can also be implemented directly in electronic hardware (by application specific integrated circuits or field programmable gate arrays for example). Addition and compare registers are only needed for the drawing process of the pixel loop. All calculations can be realized in integer values. Potentially the parameters of the curve (initial values) must possibly be precalculated by a short (micro) program.

Existing hardware accelerations like CUDA (Compute Unified Device Architecture) of Nvidia or OpenGL focus on high performance 3D rendering for parallel computing architecture. 2D Bézier curves are not part of a 3D computer scene. Such curves don't need a graphics processor to be fast enough for rasterizing and are mostly ignored by high performance libraries.

10.4 Outlook

As a future work the algorithm could be extended to rasterize cubic rational Béziers and other familiar curves.

This type of algorithm also suggests fast implementation for SIMD (Single Instruction, Multiple Data) instructions since values for x and y direction could be calculated independently.

10.5 Source code

The examples of this documents are available by public at the internet. The web address is: http://members.chello.at/easyfilter/bresenham.html

The programs have no copyright and could be used and modified by anyone as wanted.

The source code was carefully tested but are given without warranty of any kind. Use it at your own risk.

Bibliography

[Bézier, 1986] Pierre Étienne Bezier: The Mathematical Basis of the UNISURF CAD System; Butterworths, London, 1986.

[Bézout, 1764] Étienne Bézout: Recherches sur les degrés des équations résultantes de l'évanouissement des inconnues et sur les moyens qu'il convient d'employer pour trouver ces équations; Histoire de l'Académie Royale des Sciences, 1764.

[Bresenham, 1965] Jack E. Bresenham: Algorithms of Computer Control of a Digital Plotter; IBM System Journal, 1965.

[Casteljau, 1963] Paul de Faget de Casteljau: Courbes et surfaces à pôles; Technical report, Citroën, Paris, 1963.

[Cayley, 1857] Arthur Cayley: Note sur la méthode d'élimination de Bezout; Reine Angewandte Mathematik Bd. 53, 366-367, 1857.

[Emeliyanenko, 2007] Pavel Emeliyanenko: Visualization of Points and Segments of Real Algebraic Plane Curves; Master Thesis, Universität des Saarlandes, Max-Planck-Insitut für Informatik, 2007.

[Foley, 1995] James David Foley: Computer Graphics, Principles and Practice in C; Addison-Wesley Professional; 2nd edition,1995.

[Goldman et al., 1985] Ronald N. Goldman, Thomas W. Sederberg: Some applications of resultants to problems in computational geometry; The Visual Computer, Volume 1, Number 2, 101-107, 1985.

[Golipour-Koujali, 2005] M. Golipour-Koujali: General Rendering and Antialiasing Algorithms for Conic Sections; London South Bank University, 2005.

[Loop et al., 2005] Charles Loop, James F. Blinn: Resolution Independent Curve Rendering using Programmable Graphics Hardware; Association for Computing Machinery Inc., Microsoft Research, 2005.

[Marsh, 2005] Duncan Marsh: Applied Geometry for Computer Graphics and CAD; Springer, 2005.

[Piegl et al., 1996] Les Piegl, Wayne Tiller: The NURBS Book; Springer, 1996.

[Ray et al., 2011] Kumar S. Ray, Bimal Kumar Ray: A Method of Deviation for Drawing Implicit Curves; International Journal of Computer Graphics, Vol. 2, No. 2, November 2011.

[Sederberg, 2011] Thomas W. Sederberg: Computer Aided Geometric Design Course Notes; Brigham Young University, http://tom.cs.byu.edu/~557/, 2011.

[Shmakov, 2011] Sergei L. Shmakov: A Universal Method of Solving Quartic Equations; International Journal of Pure and Applied Mathematics, Volume 71 No. 2, 251-259.

[Taubin, 1994] Gabriel Taubin: Distance Approximations for Rasterizing Implicit Curves; IBM Watson Research Center, 1994.

[Thomas, 1949] Llewellyn Hilleth Thomas: Elliptic problems in linear difference equations over a network; Watson Science Computer Laboratory Report, Columbia University, New York, 1949.

[Xiaolin, 1991] Xiaolin Wu: An Efficient Antialiasing Technique; University of Western Ontario, 1991.

[Yang et al., 2000] Hongji Yang, Yang Li, Yong Kui Liu: A Pixel Level Algorithm for Drawing Curves; Proceedings of the 6th Chinese Automation and Computer Science Conference in the UK, Loughborough, 2000.

List of figures

List of programs

List of equations