Fast Algorithms For Rendering Cubic Curves

Benjamin Watson
Larry F. Hodges
Graphics, Visualization, and Usability Center
College Of Computing
Georgia Institute Of Technology
Atlanta, GA 30332

Abstract

We present two integer-only algorithms to be used in tandem for rendering cubic functions and parametric cubic curves with rational coefficients. Analysis of execution speed of existing algorithms shows that our algorithms will match or outperform other current algorithms. Furthermore, while other existing algorithms can only handle curves shaped by rational coefficients by introducing some approximation error, our algorithms always choose the best approximation. Curves may have to be split before rendering, because each algorithm only handles curves with slope in a certain range. When plotting parametric curves, our algorithms may require more bits of representation for some integer variables than other existing algorithms.

Keywords: display algorithms, curve representations, parametric curves, raster graphics

1. Introduction

Computer scientists have been developing line and curve-rendering algorithms for over 25 years. Only recently, however, have efficient algorithms for the plotting of cubic curves begun to appear. This paper will propose and develop two fast, integer-only algorithms that can be used in tandem to render cubic curves with rational coefficients defined by the function

$$y = (A_n/A_d)x^3 + (B_n/B_d)x^2 + (C_n/C_d)x + (D_n/D_d). \quad (1)$$

The algorithms are based on the midpoint method, described by Van Aken and Novak in [17] and below.

2. History And Existing Algorithms

J.E. Bresenham was the first to present a fast, integer-only line rendering algorithm in 1965 [1]. Research in line rendering since has seized on the periodic patterns shown by Bresenham's algorithm when viewed on a raster display as a means of improving algorithm speed [4,14].

Algorithms for rendering circles began to appear in the 1970s. Bresenham [2,3], Horn [7], and McIlroy [12] have all presented algorithms. Later, algorithms for rendering ellipses were published [9,15,16], and more recently, algorithms for the plotting of parabolas and hyperbolas were presented [13,15,18].

Algorithms for the rendering of cubic curves have only begun to appear in the last few years. In [11], Klassen presented two algorithms for rendering parametric cubic curves. First he identified the family of Bezier curves that are "worst-case", meaning that they are most likely to cause overflow during calculation. If $2h$ is screen length or width, Klassen asserted that "worst case" curves would have the four Bezier control points $[-h,5h,-5h,h]$ in at least one dimension, which would describe the one-dimensional parametric Bezier cubic $32ht^3 - 48ht^2 + 18ht - h$. Klassen called such curves S curves. Klassen then presented his two algorithms and outlined their relative speed and overflow restrictions for worst-case curves. Algorithm A uses a fixed-point representation of curve coordinates, and thus incorporates an inherent level of error. However, it is fast and has a liberal overflow restriction. Algorithm B divides forward differences into integer and fractional parts, providing perfect accuracy. But it is slower than algorithm A, and can only take 1024 parametric steps if overflow is to be avoided with 32-bit words. Both algorithms allow arbitrary step size and do not restrict curve segments to certain slope octants. Both can be used with non-integer coefficients. However, use of such coefficients with algorithm B would eliminate its perfect accuracy.

In [10], Klassen studied the use of these two algorithms with cubic spline curves. He envisioned the use of the algorithms with adaptive forward differencing [6,8], which dynamically adjusts step size as a curve is plotted.

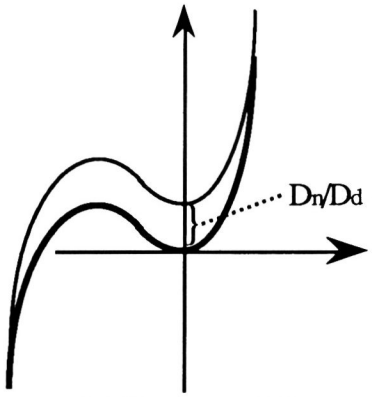

Figure 1: Elimination of the constant term can be compensated for by a translation.

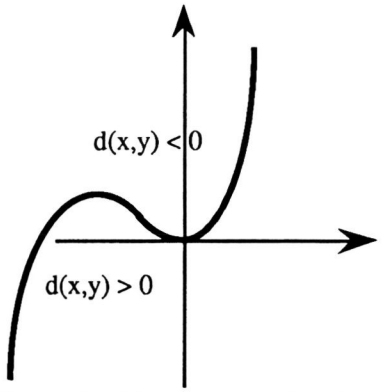

Figure 2: If the decison function is evaluated at a point above the curve, it is negative. Otherwise it is positive.

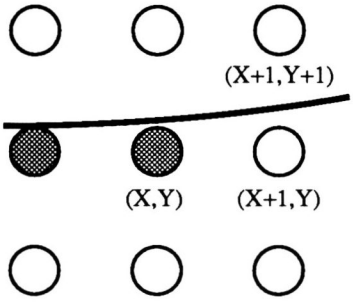

Figure 3: If |slope| < 0 and X and Y plotting directions are positive the candidate points are (X+1,Y) and (X+1, Y+1).

Table 1. The candidate and midpoints used depend on curve slope and X and Y plot direction. Points are listed in clockwise order.

Y Plot Dir	X Plot Dir	|Slope|	Candidate Points	Midpoints
+	+	< 1	(X+1,Y); (X+1,Y+1)	(X+1,Y+1/2)
+	+	>= 1	(X+1,Y+1); (X,Y+1)	(X+1/2,Y+1)
+	-	>= 1	(X,Y+1); (X-1,Y+1)	(X-1/2,Y+1)
+	-	< 1	(X-1,Y+1); (X-1,Y)	(X-1,Y+1/2)
-	+	< 1	(X-1,Y); (X-1,Y-1)	(X-1,Y-1/2)
-	+	>= 1	(X-1,Y-1); (X,Y-1)	(X-1/2,Y-1)
-	-	>= 1	(X,Y-1); (X+1,Y-1)	(X+1/2,Y-1)
-	-	< 1	(X+1,Y-1); (X+1,Y)	(X+1,Y-1/2)

Simultaneous to Klassen, Chang et al. [5] developed an algorithm similar to Klassen's algorithm B that also could be used with adaptive forward differencing. Differences between the two algorithms are minor.

3. Preliminaries

3.1. Elimination Of The Constant Term D_n/D_d

Since the last term (D_n/D_d) in (1) does not change the shape of the curve, we can render the curve described by instead rendering the curve

$$y = (A_n/A_d)x^3 + (B_n/B_d)x^2 + (C_n/C_d)x \qquad (2)$$

with a compensating translation (see figure 1). Note that if the (D_n/D_d) rational coefficient is not an integer, translation of the Y coordinate at plotting by round(D_n/D_d) alone will not necessarily produce the best approximation of the curve. In section 4 we will discuss a method of compensating for this inaccuracy.

3.2. The Midpoint Method

The midpoint method, described by Van Aken and Novak in [17], requires the incremental evaluation of a decision function that indicates which of two candidate pixels should be chosen for rendering. If the equation for a curve is $y = f(x)$, then the decision function has the form $d(x,y) = f(x) - y$. Notice that this function will have a different sign on each side of the curve f(x) (see figure 2).

Where do we evaluate this function? This depends on the slope of the curve. If $-1 < f'(x) < 1$ and we are plotting in positive X and Y directions, then if we have just plotted the point (X,Y), the two candidate points for plotting are (X+1,Y) and (X+1,Y+1). (See figure 3. Table 1 shows a complete list of candidate points.) The midpoint method evaluates the decision function at the midpoint between the candidate pixels. In our example, this midpoint is (X+1,Y+1/2), and thus we evaluate d(X+1,Y+1/2). (See figure 4. Table 1 shows a complete list of midpoints.) We

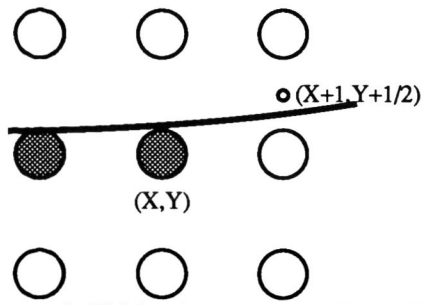

Figure 4: If |slope| < 0 and the X and Y plotting directions are positive, the midpoint is (X+1,Y+1/2).

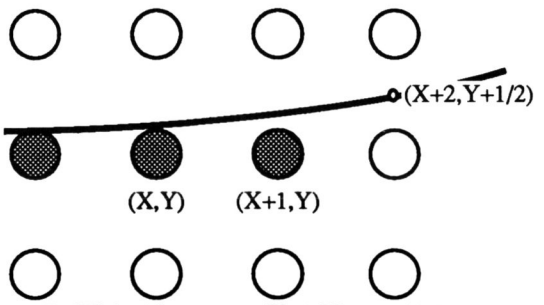

Figure 5: With this curve, (X+1,Y) would be plotted, and (X+2,Y+1/2) used as the next midpoint.

will call the decision function the "decision variable" when it is evaluated at a midpoint.

Since the sign of the decision function d(x,y) corresponds to a specific side of f(x), the sign of decision variable d(X+1,Y+1/2) indicates the side of f(x) on which the midpoint lies, and also which of the candidate pixels lies closer to the curve being plotted, f(x). In figure 4, the sign of the decision variable is negative, so the lower candidate pixel is chosen for plotting.

Once the next pixel is chosen and plotted, the decision function must be evaluated at the next midpoint to allow the plotting of the next pixel. In our example (figure 5), the appropriate decision variable would be d(X+2,Y+1/2).

3.3 Forward Differencing

Simple evaluation of the decision function at each successive midpoint would be computationally expensive. Fortunately, there is a method of incremental function evaluation, called forward differencing, which is uniquely suited to our needs. This method, which was known to Newton, involves the initialization of several difference values that may be added together to produce the value of a function at a certain point. These difference values are then themselves incrementally evaluated, to prepare for the next evaluation of the original function. Note that mulitiplication is only required for function and difference initialization. Furthermore, if all function coefficients are integers, no floating point addition is required.

As an example, consider the simple function $f(x) = 2x + 1$. $f(x+1)$ differs from $f(x)$ only by the constant difference value 2. By successively adding 2 to an initial value for $f(x)$, we could incrementally calculate the value of $f(x)$ at integer intervals on X. For the higher-order function $g(x) = x^2$, the binomial expansion $g(x+1) = (x+1)^2 = x^2 + 2x + 1$ gives us the first-order difference value $2x + 1$ for an integer interval. Since this difference value is also dependent on X, it must also be subjected to forward differencing, as already discussed. Thus the incremental calculation of $g(x) = x^2$ would require two additions per integer interval.

4. The RunRise Algorithm

Let us first find the decision function d(x,y) for equation (2) when $-1 < f'(x) < 1$. In this case, the X component of f'(x) is larger than the Y component: the curve "runs" faster than it "rises." We will label segments of f(x) where this condition holds true "RunRise." Since we will only make use of the sign of our decision function, we multiply our cubic function (2) by $2A_dB_dC_d$ to increase efficiency by eliminating the floating point division calculations. To conserve space, we use the shorthand $A_i = A_nB_dC_d$, $B_i = B_nA_dC_d$, $C_i = C_nA_dB_d$, and $D_i = A_dB_dC_d$, in the rest of this paper and the equation below. We assume without loss of generality that D_i (and thus the denominators A_d, B_d, and C_d) are positive:

$$2D_iy = 2A_ix^3 + 2B_ix^2 + 2C_ix. \qquad (3)$$

We will find it useful to plot in both positive and negative X and Y directions. Our direction-flexible decision variable $d(x\pm1,y\pm1/2)$ is then

$$2A_i(x\pm1)^3 + 2B_i(x\pm1)^2 + 2C_i(x\pm1) - 2D_i(y\pm1/2) \qquad (4)$$

where $\pm$ is positive if we are plotting in a positive direction, negative otherwise.

We must evaluate (4) incrementally as we plot the RunRise portion of f(x). To avoid computationally complex multiplications, we will use forward differencing. The difference constant d_{0y} is the difference between d(x,y) evaluated at the "current" Y coordinate, and d(x,y) evaluated at the "next" Y coordinate:

$$\begin{aligned} d_{0y} &= d(x\pm1,y\pm3/2) - d(x\pm1,y\pm1/2) \\ &= 2D_i(y\pm3/2) - 2D_i(y\pm1/2) \\ &= \pm2D_i. \end{aligned}$$

This difference constant is used to update the decision variable when a "Y step" is made -- that is, when the last plotted pixel differs from the previously plotted one in its Y coordinate.

Because the decision variable (4) is a third-order function in X, updating it when an X step is made is more complex. The second order difference function $d_2(x)$ is the difference between the decision variable at the current X coordinate and the next X coordinate:

$$\begin{aligned} d_2(x) &= d(x\pm1,y\pm1/2) - d(x,y\pm1/2) \\ &= 2A_i(x\pm1)^3 + 2B_i(x\pm1)^2 + 2C_i(x\pm1) - 2A_ix^3 \\ &\quad - 2B_ix^2 - 2C_ix \\ &= 2A_i(x^3\pm3x^2+3x\pm1) + 2B_i(x^2\pm2x+1) + 2C_i(x\pm1) \\ &\quad - 2A_ix^3 - 2B_ix^2 - 2C_ix \\ &= 2A_i(\pm3x^2+3x\pm1) + 2B_i(\pm2x+1) \pm 2C_i \\ &= \pm6A_ix^2 + (6A_i\pm4B_i)x + (\pm2A_i+2B_i\pm2C_i). \end{aligned}$$

$d_2(x)$ must itself be subjected to forward differencing. The first order difference function $d_1(x)$ is the difference between the value of $d_2(x)$ at the current and next X coordinates:

$$\begin{aligned} d_1(x) &= d_2(x\pm1) - d_2(x) \\ &= \pm6A_i(x\pm1)^2 + (6A_i\pm4B_i)(x\pm1) - \pm6kA_ix^2 \\ &\quad - (6A_i\pm4B_i)x \\ &= \pm6A_i(x^2\pm2x+1) + (6A_i\pm4B_i)(x\pm1) - \pm6A_ix^2 \\ &\quad - (6A_i\pm4B_i)x \\ &= \pm6A_i(\pm2x+1) + (\pm6A_i+4B_i) \\ &= 12A_ix + (\pm12A_i+4B_i). \end{aligned}$$

Finally, $d_1(x)$ must be subjected to forward differencing. The difference between $d_1(x)$ evaluated at the current and next X coordinates gives us constant d_{0x}:

$$\begin{aligned} d_{0x} &= d_1(x\pm1) - d_1(x) \\ &= 12A_i(x\pm1) - 12A_ix \\ &= \pm12A_i. \end{aligned}$$

Stepping one pixel at a time, and using the global declarations below, we can construct a direction-flexible algorithm for plotting the RunRise segments of cubic curves. Figures 6 and 7 show the core of this algorithm in C. For brevity's sake, we have removed the variable and function declarations (all variables are long integers). The initialization of A_i, B_i, C_i and D_i is not shown -- it is common to all curve segments.

In its loop, the RunRise algorithm performs 4 additions for each step in X, 2 additions for each step in Y. This gives a cost of

$$4a|\texttt{XEnd-X}| + 2a|\texttt{YEnd-Y}|,$$

where (X,Y) and (XEnd,YEnd) are the endpoints of the plotted curve segment, and a the cost of one addition operation.

Earlier, we noted that compensating for the elimination of the non-integer constant with translation will not necessarily produce the best approximation of the curve. Adding an appropriately signed round(D_i * ((D_n/D_d) mod 1)) to the initial decision variable will simulate a fractional Y step and improve accuracy.

5. The RiseRun Algorithm

Now let us find the decision variable needed when $|f'(x)| > 1$. In this case, it is the Y component of $f'(x)$ that is larger, so the curve will "rise" faster than it "runs." We will label segments of $f(x)$ where this condition holds true "RiseRun."

As is clear from table 1, the direction-flexible midpoint decision variable is $d(x\pm1/2,y\pm1)$. Expanded, this is

$$8A_i(x\pm1/2)^3 + 8B_i(x\pm1/2)^2 + 8C_i(x\pm1/2) - 8D_i(y\pm1). \quad (5)$$

We have used the constant $8D_i$ rather than $2D_i$ in (5) to allow the integer performance of the half step $x\pm1/2$. (5) reduces as follows:

$$\begin{aligned} &= 8A_i(x^3\pm3/2x^2+3/4x\pm1/8) + 8B_i(x^2\pm x+1/4) \\ &\quad + 8C_i(x\pm1/2) - 8D_i(y\pm1) \\ &= 8A_ix^3 + (\pm12A_i+8B_i)x^2 + (6A_i\pm8B_i+8C_i)x \\ &\quad + (\pm A_i+2B_i\pm4C_i) - 8D_i(y\pm1) \end{aligned}$$

We use this all-integer equation to initialize our decision variable. The following forward stepping increments allow us to update that decision variable:

$$\begin{aligned} d_2(x) &= d(x\pm3/2,y\pm1) - d(x\pm1/2,y\pm1) \\ &= 8A_i(x\pm3/2)^3 + 8B_i(x\pm3/2)^2 + 8C_i(x\pm3/2) \\ &\quad - 8A_i(x\pm1/2)^3 - 8B_i(x\pm1/2)^2 - 8C_i(x\pm1/2) \\ &= 8A_i(x^3\pm9/2x^2+27/4x\pm27/8) + 8B_i(x^2\pm3x+9/4) \\ &\quad + 8C_i(x\pm3/2) \\ &= 8A_i(x^3\pm3/2x^2+3/4x\pm1/8) - 8B_i(x^2\pm x+1/4) \\ &\quad - 8C_i(x\pm1/2) \\ &= \pm24A_ix^2 + (48A_i\pm16B_i)x + (\pm26A_i+16B_i\pm8C_i) \end{aligned}$$

$$\begin{aligned} d_1(x) &= d_2(x\pm1) - d_2(x) \\ &= \pm24A_i(x\pm1)^2 + (48A_i\pm16B_i)(x\pm1) - \pm24A_ix^2 \\ &\quad - (48A_i\pm16B_i)x \\ &= \pm24A_i(x^2\pm2x+1) + (48A_i\pm16B_i)(x\pm1) - \pm24A_ix^2 \\ &\quad - (48A_i\pm16B_i)x \\ &= \pm24A_i(\pm2x+1) + (\pm48A_i+16B_i) \\ &= 48A_ix + (\pm72A_i+16B_i) \end{aligned}$$

```
XDec2Const1 = Ai<<1;                                /* Used to init 2nd order diffc function
*/
XDec0Const = (XDec2Const1<<2) + (XDec2Const1<<1); /* Used to init diffc constant */
XDec2Const2 = Bi<<1;                                /* Used to init 2nd order diffc function
*/
XDec1Const = XDec2Const2<<1;                        /* Used to init 1st order diffc function
*/
XDec2Const3 = Ci<<1;                                /* Used to init 2nd order diffc function
*/
DecConst = Di;                                      /* Used to init decision variable */
YDecConst = DecConst<<1;                            /* Difference constant for a Y step */

Temp1 = XDec2Const1*X;                              /* Used several times to save multiplies
*/
if (X < XEnd) {                                     /* If plotting in positive X direction */
   XStep = 1;                                       /* Set X increment */
   XDec0 = XDec0Const;                              /* Difference constant for an X step */
   XDec1 = (Temp1<<2) + (Temp1<<1)                  /* Init 1st order diffc function */
      + XDec0Const + XDec1Const;
   XDec2 = ((Temp1<<1) + Temp1                      /* Init 2nd order diffc function */
      + (XDec0Const>>1) + XDec1Const)*X
      + XDec2Const1 + XDec2Const2 + XDec2Const3;
} else {                                            /* If plotting in negative X direction */
   XStep = -1;                                      /* Set X increment */
   XDec0 = -XDec0Const;                             /* Diffc constant for an X step */
   XDec1 = (Temp1<<2) + (Temp1<<1)                  /* Init 1st order diffc function */
      - XDec0Const + XDec1Const;
   XDec2 = (-(Temp1<<1) - Temp1                     /* Init 2nd order diffc function */
      + (XDec0Const>>1) - XDec1Const)*X
      - XDec2Const1 + XDec2Const2 - XDec2Const3;
} /* end if */

Dec = ((Temp1 + XDec2Const2)*X + XDec2Const3)*X;    /* Init decision variable */
if (Y < YEnd) {                                     /* If plotting in positive Y direction */
   YStep = 1;                                       /* Set Y increment */
   Dec = Dec + XDec2 - YDecConst*Y - DecConst;      /* Final decision variable init'zation */
} else {                                            /* If plotting in negative Y direction */
   YStep = -1;                                      /* Set Y increment */
   XDec0 = -XDec0;                                  /* Negate X differences */
   XDec1 = -XDec1;
   XDec2 = -XDec2;
   Dec = -Dec + XDec2 + YDecConst*Y - DecConst;     /* Final decision variable init'zation */
} /* end if */
```

Figure 6: RunRise algorithm initialization. (X,Y) and (XEnd,YEnd) are segment endpoints.

```
if (X == XEnd)                                      /* If degenerate curve, */
   {LinPlot(X,Y,XEnd,YEnd);                         /* Plot a line */
else                                                /* Otherwise, */
   for (X=X; X<=XEnd; X=X+XStep) {                  /* For each X in the curve */
      Plot(X,Y);                                    /* Plot a point */
      XDec2 = XDec2 + XDec1;                        /* Update the 2nd order diffc */
      XDec1 = XDec1 + XDec0;                        /* Update the 2st order diffc */
      if (Dec > 0) { /* perform Y step */           /* If must perform Y step */
         Y = Y + YStep;                             /* Adjust Y accordingly */
         Dec = Dec + XDec2 - YDecConst;             /* Update the decision var accordingly */
      } else Dec = Dec + XDec2;                     /* If no Y step, update dec var accdgly */
   } /* end for */
```

Figure 7: RunRise algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.

$$d_{0x} = d_1(x\pm1) - d_1(x)$$
$$= 48A_i(x\pm1) - 48A_ix$$
$$= \pm48A_i$$

$$d_{0y} = 8D_if(x\pm1/2,y\pm2) - 8D_if(x\pm1/2,y\pm1)$$
$$= 8D_i(y\pm2) - 8D_i(y\pm1)$$
$$= \pm8D_i.$$

Figures 8 and 9 show the RiseRun algorithm. Again, variable declarations and global initializations are not shown.

The RiseRun algorithm, like the RunRise algorithm, performs 4 additions for each step in X, 2 additions for each step in Y. Thus algorithm cost is

$$4a|\texttt{XEnd-X}| + 2a|\texttt{YEnd-Y}|.$$

Note, however, that |YEnd-Y| will in this case always be greater than |XEnd-X|.

6. Use Of The Algorithms In Tandem

Because each of the RunRise and RiseRun algorithms only function for curve segments that have slope within a certain range, plotting a cubic curve with these algorithms will typically require that the curve be split into segments which satisify the algorithms' slope range requirements. Algorithm initialization must then be performed for each curve segment, increasing overhead.

The actual curve splitting itself will also increase overhead. Since each algorithm takes as input only function constants and segment endpoints, splitting essentially involves the location of appropriate segment endpoints. For the unoptimized algorithms, these endpoints will be the points on the curve at which the conditions $|f'(x)| = 1$ and $f'(x) = 0$ hold true. Locating these points will involve floating point arithmetic. The appropriate algorithm must then be called for each segment. In the worst case, a curve will have to be split into seven such segments. However, use of the algorithms with spline and Bezier curves would typically require the splitting of curves into only two or three segments.

7. Overflow

Care must be taken to avoid overflow when using these algorithms. Below, we provide overflow analysis for the initialization and looping portions of each of the algorithms.

We begin with the RunRise algorithm. During initialization, the largest intermediate value that must be handled is the first initialization of the decision variable, `((Temp1 + XDec2Const2)*X + XDec2Const3)*X`. This represents $2A_ix^3 + 2B_ix^2 + 2C_ix$. For ease of algorithm use and analysis, we define i such that each of the coefficients $|A_i|$, $|B_i|$, $|C_i|$ and $|D_i|$ are less than i. The screen space variable $|x|$ has the maximum value $w/2$, where w is screen width. Thus in the worst case, the decision variable will take on the intermediate value $|iw(w^2/4 + w/2+ 1)|$. To represent this value without overflow, the condition

$$|iw(w^2/4 + w/2+ 1)| < 2^{bits-1},$$

where *bits* is the number of bits available for representation, must hold true. As an example, it is reasonable to expect screen width w to be less than 1280. We then have

$$1280i\,(320^2 + 640+ 1) < 2^{bits-1}$$
$$1280i\,(103041) < 2^{bits-1}$$
$$|i| < 2^{bits-1}/131892480.$$

If a 32-bit word is to be used for representation, *bits* = 32 and $|i|$ must be less than or equal to 16. Clearly, this is too restrictive. If instead 64 bits are used during initialization, we have $|i| < 3.496549627e+10$. Considering that $\lfloor \log(3.496549627e+10) \rfloor$ is 35, this is quite reasonable.

The algorithm changes the state of 5 variables while looping: `X`, `Y`, `Dec`, `XDec1`, and `XDec2`. `X` and `Y` are screen space variables, and thus will not overflow unless any screen coordinate is larger than 2^{bits-1} in magnitude -- an unlikely event, given the present state of raster technology. `XDec1` represents the difference function $d_1(x) = d_2(x\pm1) - d_2(x)$, and `XDec2` the difference function $d_2(x)$. By definition, then, `XDec2` will always be larger than `XDec1` in magnitude.

`Dec` represents the decision function d(x,y) evaluated at the midpoint $(X\pm1, Y\pm1/2)$. When curve slope $|f'(x)| < 1$ (the RunRise case), the midpoint method guarantees that this point will be a distance d of at most 1/2 in Y from the point $(X\pm1, f(X\pm1))$ (see again figure 4 and table 1). Thus we have

$$d = |f(X\pm1) - (Y\pm1/2)| \le 1/2.$$

Scaling by $2D_i$ gives

$$2D_id = |2D_if(X\pm1) - 2D_i(Y\pm1/2)| \le D_i.$$

Note now that $2D_if(X\pm1) - 2D_i(Y\pm1/2)$ is equivalent to the decision variable (4). Thus we have $|d(x\pm1,y\pm1/2)| = |\texttt{Dec}| \le D_i$.

`XDec2` represents the difference function $d_2(x) = d(x\pm1,y\pm1/2) - d(x,y\pm1/2) = g(x\pm1) - g(x)$, where g(x) is the function (3). We define the difference $d_2'(x) = d_2(x)/2D_i = f(x\pm1) - f(x)$, where f(x) is the function (2). In the RunRise case, $|f'(x)| < 1$, which implies that $d_2'(x) = |f(x\pm1) - f(x)| \le 1$, and then it follows that $|\texttt{XDec2}| = |d_2(x)| \le 2D_i$. This allows us to conclude that representing the RunRise looping variables requires only the fulfillment of the almost trivial inequality

```
XDec0Const = (Ai<<5) + (Ai<<4);                  /* Used to init diffc constant */
XDec2Const = XDec0Const>>1;                      /* Used to init 2nd order diffc fction */
XDec1Const = XDec2Const + (Bi<<4);               /* Used to init 1st order diffc fction */
YDecConst = Di<<3;                               /* Difference constant for a Y step */
XDecConst = Ci<<3;                               /* Used to init 1st/2nd ord diffc fcts */

Temp1 = (Ai<<3)*X;                               /* Used to save multiplies */
if (X < XEnd) {                                  /* If plotting in positive X direction */
   XStep = 1;                                    /* Set X increment */
   XDec0 = XDec0Const;                           /* Difference constant for an X step */
   XDec1 = (Temp1<<2) + (Temp1<<1)               /* Init 1st order diffc function */
      + XDec0Const + XDec1Const;
   XDec2 = ((Temp1<<1) + Temp1                   /* Init 2nd order diffc function */
      + XDec1Const + XDec2Const)*X
      + XDec1Const + (Ai<<1) + XDecConst;
   Dec = ((Temp1 + (XDec1Const>>1))*X            /* Init decision variable */
      + (XDec1Const>>1) - (XDec2Const>>2)
      + XDecConst)*X + Ai + (Bi<<1)
      + (Ci<<2);
} else {                                         /* If plotting in negative X direction */
   XStep = -1;                                   /* Set X increment */
   XDec0 = -XDec0Const;                          /* Diffc constant for an X step */
   XDec1 = (Temp1<<2) + (Temp1<<1)               /* Init 1st order diffc function */
      - (XDec0Const<<1) + XDec1Const;
   XDec2 = (-(Temp1<<1) - Temp1                  /* Init 2nd order diffc function */
      + XDec0Const + XDec2Const
      - XDec1Const)*X + XDec1Const
      - XDec0Const - (Ai<<1) - XDecConst;
   Dec = ((Temp1 + (XDec1Const>>1)               /* Init decision variable */
      - XDec2Const)*X - (XDec1Const>>1)
      + XDec2Const - (XDec2Const>>2)
      + XDecConst)*X - (Ci<<2) + (Bi<<1)
      - XDecConst;
} /* end if */

if (Y < YEnd) {                                  /* If plotting in positive Y direction */
   YStep = 1;                                    /* Set Y increment */
   Dec = Dec - YDecConst*Y - YDecConst;          /* Final decision variable init'zation */
} else {                                         /* If plotting in negative Y direction */
   YStep = -1;                                   /* Set Y increment */
   XDec0 = -XDec0;                               /* Negate X differences */
   XDec1 = -XDec1;
   XDec2 = -XDec2;
   Dec = -Dec + YDecConst*Y - YDecConst;         /* Final decision varaible init'zation */
} /* end if */
```

Figure 8: RiseRun algorithm initialization. (X,Y) and (XEnd,YEnd) are segment endpoints.

```
if (Y == YEnd)                                   /* If degenerate curve, */
   {LinPlot(X,Y,XEnd,YEnd);                      /* Plot a line */
else                                             /* Otherwise, */
   for (Y=Y; Y<=YEnd; Y=Y+YStep) {               /* For each Y in the curve */
      Plot(X,Y);                                 /* Plot a point */
      if (Dec < 0) { /* perform X step */        /* If must perform X step */
         X = X + XStep;                          /* Adjust X accordingly */
         Dec = Dec + XDec2 - YDecConst;          /* Update the dec variable accordingly */
         XDec2 = XDec2 + XDec1;                  /* Update the 2nd order diffc */
         XDec1 = XDec1 + XDec0;                  /* Update the 1st order diffc */
      } else Dec = Dec - YDecConst;              /* If no X step, update dec var accdgly */
   } /* end for */
```

Figure 9: RiseRun algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.

$$2D_i < 2^{bits-1}.$$

Then if *bits* = 32, we must have $D_i < 2^{30}$ to loop in the RunRise algorithm without overflow.

The overflow analysis of initialization for the RiseRun algorithm is similar to the RunRise initialization analysis. The largest intermediate value calculated during initialization is the partial sum of the decision variable $8A_ix^3 + (\pm 12A_i+8B_i)x^2 + (6A_i\pm 8B_i+8C_i)x + (\pm A_i+2B_i\pm 4C_i)$. Here the worst case value is $|i\,(w^3 + 5w^2 + 12w + 7)|$, giving us the inequality

$$|i\,(w^3 + 5w^2 + 12w + 7)| < 2^{bits-1}$$

if overflow is to be avoided. Since this inequality is clearly more restrictive than the inequality required for RunRise initialization, the use of more than 32 bits is appropriate.

Because the condition $|f'(x)| < 1$ does not hold for RiseRun curves, overflow analysis of the RiseRun looping section will differ significantly from the similar RunRise analysis. The unoptimized RiseRun algorithm changes the same five variables as the unoptimized RunRise algorithm. We can again conclude that the overflow restrictions required by `Dec` and `XDec2` will most seriously affect program utility.

In the RiseRun algorithm, $d_2(x)/2D_i = f(x\pm 3/2) - f(x\pm 1/2)$. But since $|f'(x)| \geq 1$, we cannot conclude that $|f(x\pm 3/2) - f(x\pm 1/2)| \leq 1$ and $|XDec2| \leq 8D_i$. Theoretically, the difference $|f(x\pm 3/2) - f(x\pm 1/2)|$ could be infinite. Clearly, however, a curve that fulfilled such a condition would have infinite slope, and f(x) would then simply describe a vertical line. In fact, any curve segment that fulfills the condition `XEnd = X` would for our purposes be a vertical line, and would be most quickly rendered by a primitive line plotting routine. Since our RiseRun algorithm captures such cases, we can guarantee that `XEnd` $\neq$ `X` and thus that $|f(x\pm 3/2) - f(x\pm 1/2)| \leq h$, where h is screen height. This allows us to conclude that $|XDec2| = d_2(x) = |8D_if(x\pm 3/2) - 8D_if(x\pm 1/2)| \leq 8D_ih$, and gives us the restriction

$$8D_ih \leq 2^{bits-1}.$$

The RiseRun decision variable $d(x\pm 1/2, y\pm 1)$, like the RunRise decision variable, is proportional to a distance. We'll call this distance $d' = f(X\pm 1/2) - (Y\pm 1)$. However, because $|f'(x)| \geq 1$ for RiseRun curves, $|d'|$ has the wider range $[0,h]$, which implies that $|d(x\pm 1/2, y\pm 1)|$ has the proportional range $[0,8D_ih]$, and that $|Dec| \leq 8D_ih$. Thus the above overflow restriction for `XDec2` also applies to `Dec`.

If *bits* = 32 and h = 1024, we have

$$2^{13} D_i < 2^{31}$$
$$D_i < 2^{18}.$$

8. Algorithm Use With Parametric Curves

While the algorithms presented in this paper were designed for the direct plotting of cubic functions, they can be used, with slight modifications, to plot parametric curves.

Typically, parametric curves are plotted by making both the X and Y coordinates functions of a parameter t, which may take on values in the range [0,1]. Our algorithms could be used to plot such curves by running them twice: once to obtain X coordinates, and once to obtain Y coordinates. In both cases, the algorithm variable `X` would contain the current value of the parameter t. The variable `Y` would contain an X or Y coordinate. Below, we present overflow analysis only for the parametric equation $x(t)$. For overflow restrictions for $y(t)$, simply substitute h for w.

Since our algorithms can only use a step size of 1, `X` cannot, like t, take on values only in the range [0,1]. Instead, we could let it take on integer values in the range $[0,n]$, where the even number n is the number of parametric steps desired. If we have the parametric equation

$$x(t) = At^3 + Bt^2 + Ct, \qquad (6)$$

t in [0,1], the equation

$$x(t') = A(t'^3/n^3) + B(t'^2/n^2) + C(t'/n),$$

t' in $[0,n]$, would describe the same coordinates. Thus the parametric version of the RunRise decision variable (4) is

$$A_i(t'\pm 1)^3 + B_in(t'\pm 1)^2 + C_in^2(t'\pm 1) - D_in^3(X\pm 1/2).$$

Note that we have not scaled the parametric decision variable by 2 because we know that n is an even number. The RunRise parametric differences are:

$$\begin{aligned} d_2(t') &= A_i(t'\pm 1)^3 + B_in(t'\pm 1)^2 + C_in^2(t'\pm 1) - A_it'^3 - B_int'^2 \\ &\quad - C_in^2t' \\ &= A_i(t'^3\pm 3t'^2+3t'\pm 1) + B_in(t'^2\pm 2t'+1) + C_in^2(t'\pm 1) \\ &\quad - A_it'^3 - B_int'^2 - C_in^2t' \\ &= A_i(\pm 3t'^2+3t'\pm 1) + B_in(\pm 2t'+1) \pm C_in^2 \\ &= \pm 3A_it'^2 + (3A_i\pm 2B_in)t' + (\pm A_i+B_in\pm C_in^2) \end{aligned}$$

$$\begin{aligned} d_1(t') &= \pm 3A_i(t'\pm 1)^2 + (3A_i\pm 2B_in)(t'\pm 1) - \pm 3A_it'^2 \\ &\quad - (3A_i\pm 2B_in)t' \\ &= \pm 3A_i(\pm 2t'+1) \pm (3A_i\pm 2B_in) \\ &= 6A_it' + (\pm 6A_i+2B_in) \end{aligned}$$

$$d_{0t'} = \pm 6A_i$$

For the parametric versions of our algorithms, we only present overflow restrictions for the looping sections. Analysis for the parametric RunRise variables `XDec2` and

`Dec` is quite similar to the analysis for the identically named non-parametric RunRise variables, and gives the overflow restriction

$$D_i n^3 < 2^{bits-1}.$$

If *bits* = 32 and $n = 256$, we have $D_i \leq 128$.

Remember that t' refers to a step number rather than an absolute screen location. This enables us to use a low value of n, 128, and still know that the algorithm will function over the entire screen. If we were to make n smaller, the upper limit for $|D_i|$ would increase. If we had to plot a curve segment that required more than n steps, we could simply split the curve into several sub-segments of a more managable size.

Since many parametric curves interpolate or are controlled by points chosen on a computer screen, it is often the case that the coefficients A, B, C, and D in (6) are integers, not rational. In such cases, A_i, B_i and C_i are equal to A_n, B_n, and C_n. Most important, however, is the observation that $D_i = 1$. In such cases, our RunRise overflow restriction above becomes

$$n^3 < 2^{bits-1}.$$

If *bits* = 32, we have $n \leq 1290$.

The parametric version of the RiseRun decision variable (5) is

$$8A_i(t'\pm 1/2)^3 + 8B_i n(t'\pm 1/2)^2 + 8C_i n^2(t'\pm 1/2) - 8D_i n^3(X\pm 1).$$

The RiseRun parametric differences are:

$$d_2(t') = \pm 24A_i t'^2 + (48A_i \pm 16B_i n)t' + (\pm 26A_i + 16B_i n \pm 8C_i n^2)$$

$$d_1(t') = 48A_i t' + (\pm 72A_i + 16B_i n)$$

$$d_{0t'} = \pm 6A_i$$

The overflow restriction for the RiseRun parametric case is:

$$8D_i n^3 w \leq 2^{bits-1}.$$

If *bits* = 32, $w = 1280$, and $|D_i| \leq 64$ we have

$$10D_i n^3 2^{16} \leq 2^{31}$$

$$n^3 < 2^{15}/10,$$

giving $n \leq 14$.

If $D_i = 1$, *bits* = 32 and $w = 1280$, we have

$$10n^3 2^{10} \leq 2^{32}$$

$$n^3 < 2^{22}/10,$$

giving $n \leq 74$.

9. Algorithm Comparison And Evaluation

In this section, we compare the RunRise and RiseRun algorithms as they would be used with parametric curves with the algorithms A and B presented by Klassen in [11]. Table 2 shows the operation costs and the overflow restrictions associated with Klassen's algorithms and our algorithms. We have assumed that a barrel shifter is available. Furthermore, while we generally follow Klassen's practice of weighing each branch of a conditional statement equally, we have made an exception for the branches in our algorithms' main loops. In particular, since the RunRise algorithm plots curve segments with absolute slope less than one, we have assumed (rather conservatively) that the algorithm will make 3 X steps for every 2 Y steps, and thus that the `if` statment in figure 7 will be true only 40% of the time. By similar logic, we assume that the `if` statment in figure 9 will be true only 40% of the time. As curve slope nears zero or infinity, these percentages decrease, and algorithm performance improves.

Clearly, the main loops of both the RunRise and RiseRun algorithms use less operations than Klassen's algorithm B, and are comparable in cost to Klassen's algorithm A. During initialization, our algorithms use no expensive divide operations, and use about half the number of add and multiply operations used by Klassen's algorithms. It should be noted, however, that while our algorithms require that a cubic curve be split into segments, both of Klassen's algorithms A and B do not depend on slope. Thus, initialization for the RunRise and RiseRun algorithms will in practice require slightly more addition and multiply operations than Klassen's algorithms, as well as several floating point calculations.

Klassen's algorithm A has by far the most liberal overflow restriction -- it is linear in n, the number of parametric steps, and w, screen width in pixels. The RunRise algorithm and Klassen's algorithm B both have restrictions that are cubic in n, with the RunRise algorithm allowing twice as many parametric steps as algorithm B. The RiseRun overflow restriction, however, is cubic in n and linear in w. This severely restricts the RiseRun algorithm's utility if only 32 bits are available.

It should be noted that the overflow restriction shown for Klassen's algorithms guarantee that *both* initialization and looping will be accomplished without overflow. The restrictions shown for the RunRise and RiseRun algorithms guarantee only that looping will be accomplished without overflow. Initialization of our algorithms requires many more bits for representation than does initialization for Klassen's algorithms (with the exception of algorithm A's extended precision divide (**xdlv**) operation).

Klassen's algorithm A uses a fixed point approach, and thus incorporates an inherent level of error not present in the other algorithms. Both of Klassen's algorithms can be used with rational coefficients, but doing so would require floating point calculation, increase error in algorithm A, and introduce error into algorithm B. Our algorithms remain perfectly accurate even with rational coefficients.

Table 2. A comparison of Klassen's algorithms from [11] with the algorithms presented in this paper as used for plotting parametric curves with integer coefficients. n is the number of parametric steps made, w is the width of the screen in pixels, *bits* is the number of bits used to represent algorithm values, Z is the number of bits of fractional precision.

	Operation Cost											
	Main Loop				Initialization							Overflow
Algorithm	+	if	<<	=	+	*	div	xdiv	<<	if	=	Restriction
RunRise	3	1	0	3	16	5	0	0	11	2	16	$n^3 < 2^{bits-1}$
RiseRun	2	1	0	2	22	5	0	0	16	2	14	$8wn^3 < 2^{bits-1}$
Klassen's A	3	0	3	3	17+3Z	12	1	5	2	Z	28+2Z	$46wn < 2^{bits-1}$
Klassen's B	10	3	0	10	40	12	5	0	0	6	33	$2n^3 < 2^{bits-1}$

If non-parametric curves are being plotted, overflow restrictions for our algorithms improve: the RunRise algorithm requires only that the product of the rational denominators D_i be less than 2^{bits-1}, and the RiseRun algorithm is similar, but is linear in screen width. Overflow restrictions for Klassen's algorithms in such a case will not show such a significant improvement.

In summary, Klassen's algorithms make efficient use of available bits, but at the price of algorithm speed or accuracy. Our algorithms require more representational bits, but are faster and more accurate. We believe that if word size is 64 or larger, or non-parametric curves are being rendered, our algorithms are clear winners.

10. Conclusions And Future Work

We have presented integer-only algorithms that allow fast, accurate plotting of cubic curves. Analysis shows that using these algorithms to plot parametric curves may require more representational bits than already existing algorithms. But if such bits are not at a premium, or non-parametric curves are being plotted, our algorithms are the algorithms of choice.

We plan to explore further the use of these algorithms with parametric curves, spline curves, and Bezier curves. In particular, we would like to explore the use of these algorithms with adaptive forward differencing.

11. References

1. Bresenham, J. E. "Algorithm for computer control of a digital plotter," IBM Syst. J. 4(1) (1965): 25-30.
2. Bresenham, J. E. "Algorithms for circular arc generation," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 197-218.
3. Bresenham, J. E. "A linear algorithm for incremental digital display of digital arcs," Commun. ACM. 20(2) (Feb. 1977): 100-106.
4. Bresenham, J. E. "Run length slice algorithm for incremental lines," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 59-104.
5. Chang, S-L., Shantz, M., and Rocchetti, R. "Rendering cubic curves and surfaces with integer adaptive forward differencing," Comput. Graph., 23(3) (Jul. 1989): 157-166.
6. Foley, J., van Dam, A., Feiner, S., and Hughes, J. Computer Graphics: Principles And Practice, Addison-Wesley, Reading, Mass., (1990).
7. Horn, B. K. P. "Circle generators for display devices," Comput. Graph. Image Process. 5 (1976): 280-288.
8. Lien, S-L., Shantz, M., and Pratt, V. "Adaptive forward differencing for rendering curves and surfaces," Comput. Graph., 21(4) (Jul. 1987): 111-118.
9. Kappel, M. R. "An ellipse-drawing algorithm for raster displays," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 257-280.
10. Klassen, R. V. "Drawing antialiased cubic spline curves," ACM Trans. Graph. 10(1) (Jan. 1991): 92-108.
11. Klassen, R. V. "Integer forward differencing of cubic polynomials," ACM Trans. Graph. 10(2) (Apr. 1991): 152-181.
12. McIlroy, M. D. "Best approximate circles on integer grids," ACM Trans. Graph. 2(4) (Oct. 1983): 237-263.
13. Metzger, R. A. "Computer generated graphics segments in a raster display," Spring 1969 Joint Computer Journal Conference, AFIPS Conf. Proc.: 161-172.
14. Pitteway, M. "The relationship between Euclid's algorithm and run length encoding," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 105-112.
15. Surany, A. P. "An ellipse-drawing algorithm for raster displays," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 281-285.
16. Van Aken, J. "An efficient ellipse-drawing algorithm," IEEE Comput. Graph. & Appl. 4(9) (Sept. 1984): 24-35.
17. Van Aken, J., and Novak, Mark. "Curve drawing algorithms for raster displays," ACM Trans. Graph. 4(2) (Apr. 1985): 147-169.
18. Watson, B., and Hodges, L. "A fast algorithm for rendering quadratic curves on raster displays," SEACM Conf. Proc. 27 (Apr. 1989): 160-165.