

# Algorithms For Rendering Cubic Curves

Benjamin Watson and 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. We then show how to take advantage of curve shape to improve algorithm performance. 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. When plotting parametric curves, our algorithms may require more bits of representation for some integer variables than other existing algorithms.

Categories and Subject Descriptors: G.1.1 [**Numerical Analysis**]: Interpolation — *spline and piecewise polynomial interpolation*; G.1.2 [**Numerical Analysis**]: Approximation — *spline and piecewise polynomial approximation*; I.3.3 [**Computer Graphics**]: Picture/Image Generation — *display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling — *curve, surface, solid, and object representations, geometric algorithms, languages and systems*.

General Terms: Algorithms

Additional Key Words And Phrases: rendering, parametric curves, raster graphics.

## 1. INTRODUCTION

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

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

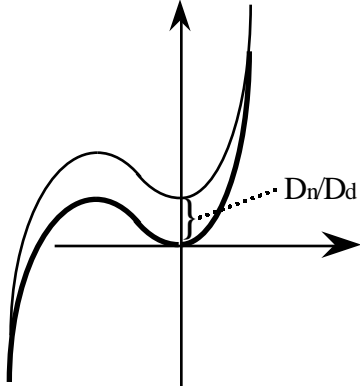
The algorithms are based the midpoint method, described by Van Aken and Novak in [17] and below.

## 2. HISTORY AND EXISTING ALGORITHMS

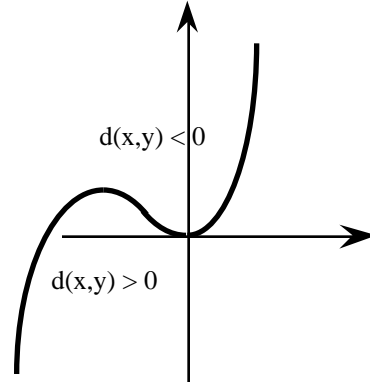
J. E. Bresenham was the first to present a fast, integer-only line rendering algorithm in 1965 [1]. Research in line rendering since has seized on the periodic patterns shown by Bresenham's algorithm when viewed on a raster display as a means of improving algorithm speed [4,14].

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



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



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

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.

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.

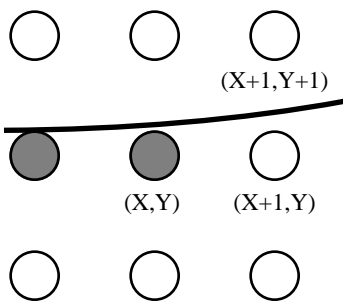
### 3. PRELIMINARIES

#### 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

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

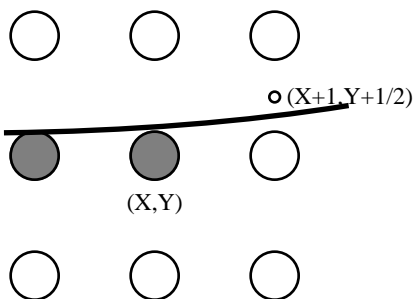
with a compensating translation (see figure 1). Note that if the ( $D_n/D_d$ ) rational coefficient is not an integer, translation of the Y coordinate at plotting by  $\text{round}(D_n/D_d)$  alone will not necessarily produce the best approximation of the curve. In section 4, we discuss a method of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



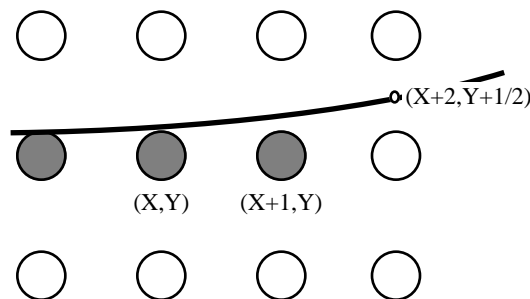
**Figure 3:** If  $|\text{slope}| < 0$  and X and Y plotting directions are positive the candidate points are  $(X+1, Y)$  and  $(X+1, Y+1)$ .

**Table 1**  
The candidate points and the midpoint used depend on curve slope and X and Y plotting directions. The points here are listed in clockwise order.

Y Plot Dir	X Plot Dir	Slope	Candidate Points	Midpoints
positive	positive	$< 1$	$(X+1, Y); (X+1, Y+1)$	$(X+1, Y+1/2)$
positive	positive	$\geq 1$	$(X+1, Y+1); (X, Y+1)$	$(X+1/2, Y+1)$
positive	negative	$\geq 1$	$(X, Y+1); (X-1, Y+1)$	$(X-1/2, Y+1)$
positive	negative	$< 1$	$(X-1, Y+1); (X-1, Y)$	$(X-1, Y+1/2)$
negative	positive	$< 1$	$(X-1, Y); (X-1, Y-1)$	$(X-1, Y-1/2)$
negative	positive	$\geq 1$	$(X-1, Y-1); (X, Y-1)$	$(X-1/2, Y-1)$
negative	negative	$\geq 1$	$(X, Y-1); (X+1, Y-1)$	$(X+1/2, Y-1)$
negative	negative	$< 1$	$(X+1, Y-1); (X+1, Y)$	$(X+1, Y-1/2)$



**Figure 4:** If  $|\text{slope}| < 0$  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.

### 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 for 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 for the complete list of midpoints). We 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 multiplication 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:

$$2D_i y = 2A_i x^3 + 2B_i x^2 + 2C_i x. \quad (3)$$

We will find it useful to plot in both positive and negative X and Y directions. Our direction-flexible decision variable is then

$$d(x \pm 1, y \pm 1/2) = 2A_i(x \pm 1)^3 + 2B_i(x \pm 1)^2 + 2C_i(x \pm 1) - 2D_i(y \pm 1/2) \quad (4)$$

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:

$$\begin{aligned} d_{0y} &= d(x \pm 1, y \pm 3/2) - d(x \pm 1, y \pm 1/2) \\ &= 2D_i(y \pm 3/2) - 2D_i(y \pm 1/2) \\ &= \pm 2D_i. \end{aligned}$$

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:

$$\begin{aligned} d_2(x) &= d(x \pm 1, y \pm 1/2) - d(x, y \pm 1/2) \\ &= 2A_i(x \pm 1)^3 + 2B_i(x \pm 1)^2 + 2C_i(x \pm 1) - 2A_i x^3 - 2B_i x^2 - 2C_i x \\ &= 2A_i(x^3 \pm 3x^2 + 3x \pm 1) + 2B_i(x^2 \pm 2x + 1) + 2C_i(x \pm 1) - 2A_i x^3 - 2B_i x^2 - 2C_i x \\ &= 2A_i(\pm 3x^2 + 3x \pm 1) + 2B_i(\pm 2x + 1) \pm 2C_i \\ &= \pm 6A_i x^2 + (6A_i \pm 4B_i)x + (\pm 2A_i + 2B_i \pm 2C_i). \end{aligned}$$

$d_2(x)$  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:

$$\begin{aligned} d_1(x) &= d_2(x \pm 1) - d_2(x) \\ &= \pm 6A_i(x \pm 1)^2 + (6A_i \pm 4B_i)(x \pm 1) - \pm 6A_i x^2 - (6A_i \pm 4B_i)x \\ &= \pm 6A_i(x^2 \pm 2x + 1) + (6A_i \pm 4B_i)(x \pm 1) - \pm 6A_i x^2 - (6A_i \pm 4B_i)x \\ &= \pm 6A_i(\pm 2x + 1) + (\pm 6A_i + 4B_i) \\ &= 12A_i x + (\pm 12A_i + 4B_i). \end{aligned}$$

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  $d_{0x}$ :

$$\begin{aligned} d_{0x} &= d_1(x+1) - d_1(x) \\ &= 12A_i(x+1) - 12A_ix \\ &= \pm 12A_i. \end{aligned}$$

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

$$4a|x_{\text{End}}-x| + 2a|y_{\text{End}}-y|,$$

where  $(x,y)$  and  $(x_{\text{End}},y_{\text{End}})$  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  $\text{round}(D_i * ((D_n/D_d) \bmod 1))$  to the initial decision variable will simulate a fractional Y step and improve accuracy.

## 5. AN OPTIMIZED RUNRISE ALGORITHM

In RunRise cubic curve segments, as  $f'(x)$  approaches zero, the number of Y steps will decrease greatly, resulting in the plotting on-screen of a series of ever-longer horizontal lines, which we will call "runs" (figure 8). This implies that the length of the next run  $L_{i+1}$  will always be greater than the length of the previous run  $L_i$ . If plotting direction was adjusted to aim at slope-zero points, algorithm speed could be improved by plotting, at each Y step, a beginning run segment of length  $L_i$  (figure 9), in effect "skipping"  $L_i$  X steps of the above, unoptimized algorithm. Normal plotting of the the remaining  $L_{i+1}-L_i$  pixels of the run could then be completed using the techniques of the above algorithm.

```

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 */

```

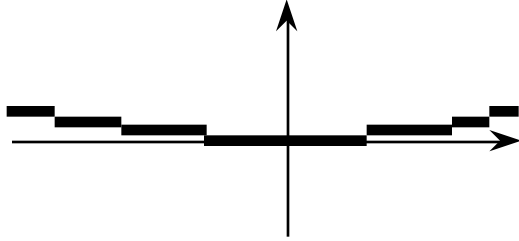
**Figure 6:** RunRise algorithm initialization. (X,Y) and (XEnd,YEnd) are segment endpoints.

```

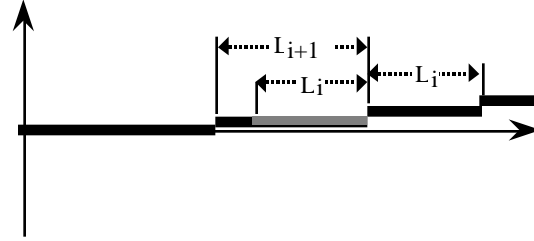
if (X == XEnd) /* If degenerate curve, */
    {LinPlot(X,Y,XEnd,YEnd); /* Plot a line */
else /* Otherwise, */
    for (X=X; X<=XEnd; X=X+XStep) { /* For each X in the curve */
        Plot(X,Y); /* Plot a point */
        XDec2 = XDec2 + XDec1; /* Update the 2nd order diffc */
        XDec1 = XDec1 + XDec0; /* Update the 2st order diffc */
        if (Dec > 0) { /* perform Y step */
            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 */

```

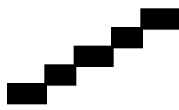
**Figure 7:** RunRise algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.



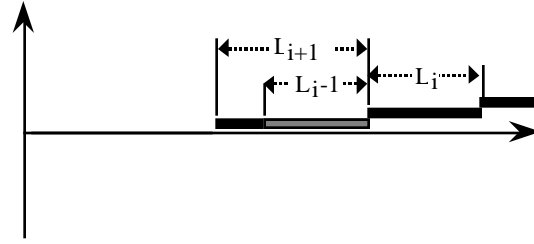
**Figure 8:** Plotted runs increase in length if plotting direction heads towards points with slope equal to zero.



**Figure 9:** Run  $i+1$  could be plotted by first plotting a run of length  $L_i$ , and then using normal midpoint method techniques.



**Figure 10:** A plotted curve segment with  $1/3 \leq \text{slope} \leq 1/2$ . Run length alternates between 2 and 3.



**Figure 11:** Run  $i+1$  is safely plotted by first plotting a run of length  $L_{i-1}$ , and then using normal midpoint method techniques.

There is one catch: run length does not increase in a strict fashion. When for an integer  $I$ , the condition  $1/I \leq f'(x) \leq 1/(I+1)$  holds over several  $Y$  steps, run length will alternate between  $I+1$  and  $I$  (figure 10). We will allow for this by assuming at each  $Y$  step that the ensuing run length  $L_{i+1}$  will only be greater than or equal to  $L_{i-1}$ , which would result in an initial run segment plot of length  $L_{i-1}$  (figure 11).

When we plot such run segments, we must at once take  $l = L_{i-1}$  forward differencing steps in our decision variable. In other words, given  $d_1(x)$ ,  $d_2(x)$  and  $d(x \pm 1, y \pm 1/2)$ , we must find  $d_1(x \pm l)$ ,  $d_2(x \pm l)$ , and  $d(x \pm (l+1), y \pm 1/2)$ . For the first-order decision function  $d_1(x)$ , we have

$$\begin{aligned} d_1(x) &= d_1(x) \\ d_1(x \pm 1) &= d_1(x) + d_{0x} \\ d_1(x \pm 2) &= d_1(x) + d_{0x} + d_{0x} \\ d_1(x \pm 3) &= d_1(x) + d_{0x} + d_{0x} + d_{0x} \\ &\text{etc.} \end{aligned}$$

Clearly, this sequence forms the sum

$$d_1(x \pm l) = d_1(x) + \sum_{i=0}^{l-1} d_{0x}$$

The closed form of this sum is

$$d_1(x \pm l) = d_1(x) \pm 12A_i l.$$

For the second order difference function  $d_2(x)$ , the sequence of values is

$$\begin{aligned} d_2(x) &= d_2(x) \\ d_2(x\pm 1) &= d_2(x) + d_1(x) \\ d_2(x\pm 2) &= d_2(x) + d_1(x) + d_1(x\pm 1) \\ d_2(x\pm 3) &= d_2(x) + d_1(x) + d_1(x\pm 1) + d_1(x\pm 2) \\ &\text{etc.} \end{aligned}$$

This gives us the sum

$$d_2(x\pm l) = d_2(x) + \sum_{i=0}^{l-1} d_1(x\pm i).$$

Below, we solve for the closed form:

$$\begin{aligned} d_2(x\pm l) &= d_2(x) + \sum_{i=0}^{l-1} (d_1(x) \pm 12A_i i) \\ &= d_2(x) + l d_1(x) + \sum_{i=1}^{l-1} \pm 12A_i i \\ &= d_2(x) + l d_1(x) + (\pm 12A_i) \sum_{i=1}^{l-1} i \\ &= d_2(x) + l d_1(x) + (\pm 12A_i) l(l-1)/2 \\ &= d_2(x) + l d_1(x) + (\pm 6A_i l) - (\pm 6A_i) \\ &= d_2(x) + l(d_1(x) + (\pm 6A_i l) - (\pm 6A_i)) \\ &= d_2(x) + l(d_1(x\pm l/2) \pm 6A_i). \end{aligned}$$

Updating the decision variable  $d(x\pm 1, y\pm 1/2)$  is the most complex. Beginning with the summation

$$d(x\pm(l+1), y\pm 1/2) = d(x\pm 1, y\pm 1/2) + \sum_{i=0}^{l-1} d_2(x\pm i),$$

we solve below for the closed form that requires the fewest multiplications and additions:

$$\begin{aligned} &= d(x\pm 1, y\pm 1/2) + \sum_{i=0}^{l-1} (d_2(x) + i d_1(x) + (\pm 12A_i) i(i-1)/2) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) \sum_{i=1}^{l-1} i + \sum_{i=1}^{l-1} ((\pm 12A_i) i(i-1)/2) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) l(l-1)/2 + (\pm 12A_i) \sum_{i=1}^{l-1} i(i-1)/2 \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) l(l-1)/2 + (\pm 6A_i) \sum_{i=1}^{l-1} i^2 - i \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) l(l-1)/2 + (\pm 6A_i) (\sum_{i=1}^{l-1} i^2 - \sum_{i=1}^{l-1} i) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) l(l-1)/2 + (\pm 6A_i) (l(l-1)(2l-1)/6 - l(l-1)/2) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + d_1(x) l(l-1)/2 + (\pm 6A_i) l(l-1)/2((2l-1)/3 - 1) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + l(l-1)/2 (d_1(x) + (\pm 6A_i) ((2l-1)/3 - 1)) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + l(l-1)/2 (d_1(x) + (\pm 2A_i) (2l-1) - (\pm 6A_i)) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + l(l-1)/2 (d_1(x) + (\pm 4A_i l) - (\pm 8A_i)) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + (l^2-l)/2 (d_1(x) + (\pm 4A_i l) - (\pm 8A_i)) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + (l^2-l)/2 d_1(x) + (l^2-l) (\pm 2A_i l) - (l^2-l)(\pm 4A_i) \\ &= d(x\pm 1, y\pm 1/2) + l d_2(x) + l^2/2 d_1(x) - l/2 d_1(x) + (\pm 2A_i l^3) - (\pm 2A_i l^2) - (\pm 4A_i l^2) - (\pm 4A_i l) \\ &= d(x\pm 1, y\pm 1/2) + l(d_2(x) + l/2 d_1(x) - 1/2 d_1(x) + (\pm 2A_i l^2) - (\pm 6A_i l) - (\pm 4A_i)) \\ &= d(x\pm 1, y\pm 1/2) + l(d_2(x) + l/2 d_1(x) - 1/2 d_1(x\pm l) + (\pm 2A_i l^2) - (\pm 4A_i)) \\ &= d(x\pm 1, y\pm 1/2) + l(d_2(x) + 1/2(-d_1(x\pm l) + l(\pm 4A_i l + d_1(x)))) - (\pm 4A_i) \end{aligned}$$

If skipped run segment length  $l = L_i - 1$  is a power of two, all the adjustments above can be performed using only left shifts and additions.

Figure 12 shows the optimized plotting algorithm. It makes use of the same global declarations of  $A_i$ ,  $B_i$ ,  $C_i$  and  $D_i$  used by the unoptimized algorithm. Again, we have removed all variable declarations. Rather than reprint all the unoptimized initialization statements, we show only the necessary additional statements. We make use of indirect function calls to conserve space. If the last iteration of the loop is partially unrolled, the indirect calls can be replaced by direct calls. Also, if the algorithm is split into two algorithms specialized for positive or negative X plotting directions, the absolute function call may be removed.

Notice that where a single step in X costs 4 additions, one skip in X costs 10 additions and 6 shifts. In machines with barrel shifters, a shift is much less expensive than an addition. A conservative estimate would give an addition 20 times the cost of a shift. Thus to break even, at least  $\lceil \log(3) \rceil = 4$  pixels must be skipped. Thus the smallest skip size is 4.

Skipping is not performed when  $|f'(x)| > 1/4$ , so cost in this case is similar to cost for the unoptimized algorithm. However, because the algorithm checks for skip size adjustments at each Y step, algorithm overhead has increased: the skipping algorithm uses 2 more additions and 1 shift for each step in Y. Thus, if the entire curve has  $|f'(x)| > 1/4$ , total cost is

$$4a|X_{\text{End}}-X| + (4a + s)|Y_{\text{End}}-Y|,$$

where  $s$  is the cost of a shift operation. When  $|f'(x)| < 1/4$  over the entire curve, skip size is checked and a skip made at each Y step. Increasing skip size by a factor of  $2^n$  costs  $a(n + 1) + sn$ . For most curves, all adjustments except the first will have  $n = 1$ , only doubling skip size. If skip size is adjusted the minimum of one time, and  $1/M$  is the largest slope on the curve segment, then total cost is

$$(14a + 7s)|Y_{\text{End}}-Y| + 4a(|X_{\text{End}}-X| - 2^{\lfloor \log(M) \rfloor} |Y_{\text{End}}-Y|).$$

If skip size is adjusted the maximum of  $|Y_{\text{End}}-Y|$  times, total cost is

$$(16a + 8s)|Y_{\text{End}}-Y| + 4a(|X_{\text{End}}-X| - (2^{\lfloor \log(M) \rfloor + |Y_{\text{End}}-Y|} - 2^{\lfloor \log(M) \rfloor})).$$

The cost of a curve segment with portions that fulfill both  $|f'(x)| < 1/4$  and  $|f'(x)| > 1/4$  will be the sum of the costs of each of the separate portions.

Obviously, the performance of the optimized algorithm depends heavily on curve slope. In general, the closer the absolute slope of a curve is to zero, the better the performance of the optimized RunRise algorithm. The optimized algorithm should not be used if the smallest absolute slope of the plotted curve is greater than  $1/4$ .

```

if (X < XEnd) /* If plotting toward positive X */
    SkipSize = 2; /* Positive skip size */
else SkipSize = -2; /* Otherwise negative skip size */
XDecSkipConst = XDec0>>1; /* Used when updating 1st/2nd ord diffc */
DecSkipConst = Ai<<2; /* Used when updating decision variable */
if (XDec0 < 0) /* Adjust for plotting direction */
    DecSkipConst = -DecSkipConst;

XLastYStep = X; /* Where last Y step was made */
CheckShift = 2; /* Range w/i which skipping begins (4) */
SkipShift = 1; /* Skip size in X dimension */
NextPlot = PtPlot; /* Set first plot type */

while (Y != YEnd) { /* While we haven't reached the last run */
    (*NextPlot)(X,Y,XLastYStep,Y); /* Perform a line or point plot */
    if (Dec <= 0) { /* If not making a Y step, don't skip */
        X += XStep; /* Increment X */
        XDec2 += XDec1; /* Increment 2nd order X diffc */
        XDec1 += XDec0; /* Increment 1st order X diffc */
        Dec += XDec2; /* Increment decision variable */
        NextPlot = PtPlot; /* Indicate next plot type */
    } else { /* adjust skip size, plot */ /* If making a Y step */
        RunHighBits = abs(X-XLastYStep)>>CheckShift; /* Find most sig bits in len last run */
        if (RunHighBits != 0) { /* If > 2*skip size, adjust size */
            NumHighBits = 1; /* How many times to double skip size */
            while (RunHighBits != 1) { /* While more doubling required */
                NumHighBits++; /* Record one more doubling */
                RunHighBits >>= 1; /* Elim one doubling from RunHighBits */
            } /* end while */
            SkipSize <=<= NumHighBits; /* Double skip size needed num times */
            SkipShift += NumHighBits; /* Change dec var skip shift to match */
            CheckShift += NumHighBits; /* Change most sig bit rng for last run */
        } /* end if */

        XLastYStep = X+XStep; /* Record location new Y step */
        Y += YStep; /* Increment Y to perform Y step */
        if (SkipShift == 1) { /* If skip size too small */
            X += XStep; /* Perform an X step as above */
            XDec2 += XDec1;
            XDec1 += XDec0;
            Dec += XDec2 - YDecConst;
            NextPlot = PtPlot;
        } else { /* If skip size lg enuf to reduce cost */
            X += SkipSize; /* Increment X by skip size */
            OldXDec1 = XDec1;
            HalfXDec1Skip = (XDecSkipConst<<SkipShift);
            XDec1 += HalfXDec1Skip;
            OldXDec2 = XDec2;
            XDec2 += ((XDec1 /* Update 2nd order diffc */
                - XDecSkipConst)<<SkipShift);
            XDec1 += HalfXDec1Skip; /* Update 1st order diffc */
            DecSkip = (OldXDec1
                + (DecSkipConst<<SkipShift))
                << SkipShift;
            DecSkip = (DecSkip - XDec1) >> 1;
            DecSkip = (OldXDec2 + DecSkip
                - DecSkipConst) << SkipShift;
            Dec += DecSkip - YDecConst; /* Update decision variable */
            NextPlot = LinPlot; /* Indicate next plot type */
        } /* end if */
    } /* end if */
} /* end while */
(*LinPlot)(XEnd,YEnd,XLastYStep,Y); /* Plot last run */

```

**Figure 12:** The optimized RunRise algorithm. (X,Y) and (XEnd,YEnd) are endpoints.

## 6. 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  $d(x\pm 1/2, y\pm 1)$ .

Expanded, this is

$$d(x\pm 1/2, y\pm 1) = 8A_i(x\pm 1/2)^3 + 8B_i(x\pm 1/2)^2 + 8C_i(x\pm 1/2) - 8D_i(y\pm 1). \quad (5)$$

We have used the constant  $8D_i$  rather than  $2D_i$  in (5) to allow the integer performance of the half step  $x\pm 1/2$ . (5) reduces as follows:

$$\begin{aligned} &= 8A_i(x^3\pm 3/2x^2+3/4x\pm 1/8) + 8B_i(x^2\pm x+1/4) + 8C_i(x\pm 1/2) - 8D_i(y\pm 1) \\ &= 8A_ix^3 + (\pm 12A_i+8B_i)x^2 + (6A_i\pm 8B_i+8C_i)x + (\pm A_i+2B_i\pm 4C_i) - 8D_i(y\pm 1) \end{aligned}$$

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

$$\begin{aligned} d_2(x) &= d(x\pm 3/2, y\pm 1) - d(x\pm 1/2, y\pm 1) \\ &= 8A_i(x\pm 3/2)^3 + 8B_i(x\pm 3/2)^2 + 8C_i(x\pm 3/2) - 8A_i(x\pm 1/2)^3 - 8B_i(x\pm 1/2)^2 \\ &\quad - 8C_i(x\pm 1/2) \\ &= 8A_i(x^3\pm 9/2x^2+27/4x\pm 27/8) + 8B_i(x^2\pm 3x+9/4) + 8C_i(x\pm 3/2) \\ &\quad - 8A_i(x^3\pm 3/2x^2+3/4x\pm 1/8) - 8B_i(x^2\pm x+1/4) - 8C_i(x\pm 1/2) \\ &= \pm 24A_ix^2 + (48A_i\pm 16B_i)x + (\pm 26A_i+16B_i\pm 8C_i) \end{aligned}$$

$$\begin{aligned} d_1(x) &= d_2(x\pm 1) - d_2(x) \\ &= \pm 24A_i(x\pm 1)^2 + (48A_i\pm 16B_i)(x\pm 1) - \pm 24A_ix^2 - (48A_i\pm 16B_i)x \\ &= \pm 24A_i(x^2\pm 2x+1) + (48A_i\pm 16B_i)(x\pm 1) - \pm 24A_ix^2 - (48A_i\pm 16B_i)x \\ &= \pm 24A_i(\pm 2x+1) + (\pm 48A_i+16B_i) \\ &= 48A_ix + (\pm 72A_i+16B_i) \end{aligned}$$

$$\begin{aligned} d_{0x} &= d_1(x\pm 1) - d_1(x) \\ &= 48A_i(x\pm 1) - 48A_ix \\ &= \pm 48A_i \end{aligned}$$

$$\begin{aligned} d_{0y} &= 8D_if(x\pm 1/2, y\pm 2) - 8D_if(x\pm 1/2, y\pm 1) \\ &= 8D_i(y\pm 2) - 8D_i(y\pm 1) \\ &= \pm 8D_i. \end{aligned}$$

Figures 13 and 14 show the unoptimized RiseRun algorithm. Again, variable declarations and global initializations are not shown.

```

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

Temp1 = (Ai<<3)*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 */
        + 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 variable init'zation */
} /* end if */

```

**Figure 13:** 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 */
            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 14:** RiseRun algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.

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

$$4a|X_{\text{End-X}}| + 2a|Y_{\text{End-Y}}|.$$

Note, however, that  $|Y_{\text{End-Y}}|$  will in this case always be greater than  $|X_{\text{End-X}}|$ .

## 7. AN OPTIMIZED RISERUN ALGORITHM

With RiseRun curves, as  $f'(x)$  approaches infinity, the number of X steps will decrease greatly, resulting in the plotting of series of ever-longer vertical runs. We will adjust our plotting direction so that we always plot towards infinite-slope points. This allows us to skip like we did with RunRise curves. Since runs are now vertical, we skip Y steps rather than X steps. Updating the decision variable to reflect this skip is thus much easier:

$$d(x \pm 1/2, y \pm (l+1)) = d(x \pm 1/2, y \pm 1) \pm 8D_i l.$$

Figure 15 shows the optimized RiseRun skipping algorithm. Variable declarations are not included. Rather than reprint all of the unoptimized RiseRun algorithm's initializations, we show only the needed additional initializations. Unrolling the last iteration would allow removal of the indirect function calls. If the algorithm is split into two algorithms specialized for positive or negative Y plotting directions, the absolute function call may be removed.

A Y step in this algorithm costs 2 additions, while a skip costs 2 additions plus 1 shift. Thus a skip size of 2 is adequate for a net gain. Thus skipping is not performed when  $|f'(x)| < 2$ , so cost in this case is again similar to cost for our non-skipping RiseRun algorithm. The skipping algorithm uses 2 more additions and 1 shift for each step in X. Thus, if the entire curve has  $|f'(x)| < 2$ , total cost is

$$2a|Y_{\text{End-Y}}| + (6a + s)|X_{\text{End-X}}|.$$

When  $|f'(x)| > 2$  over the entire curve, we skip at each X step. Adjusting the skip size requires 2 additions and one shift (again, normally only the initial adjustment will be more costly). If skip size is adjusted the minimum of one time, and  $m$  is the smallest slope on the curve, total cost is

$$(6a + s)|X_{\text{End-X}}| + 2a(|Y_{\text{End-Y}}| - 2^{\lfloor \log(m) \rfloor} |X_{\text{End-X}}|).$$

```

if (Y < YEnd)                               /* If plotting toward positive Y */
    SkipSize = 1;                             /* Positive skip size */
else SkipSize = -1;                           /* Otherwise negative skip size */
YLastXStep = Y;                               /* Where last X step was made */
YDecSkipConst = YDecConst;                   /* Used when updating decision variable */
CheckShift = 1;                              /* Range w/i which skipping begins (2) */
NextPlot = PtPlot;                           /* Set first plot type */

while (X != XEnd) {                           /* While we haven't reached the last run */
    (*NextPlot)(X,Y,X,YLastXStep);           /* Perform a line or point plot */
    if (Dec >= 0) { /* no X step */          /* If not making an X step, don't skip */
        Y += YStep;                          /* Increment Y */
        Dec -= YDecConst;                    /* Update decision variable */
        NextPlot = PtPlot;                  /* Indicate next plot type */
    } else { /* adjust skip size, plot */    /* If making an X step */
        RunHighBits = abs(Y-YLastXStep)>>CheckShift; /* Find most sig bits in len last run */
        if (RunHighBits != 0) {             /* If > 2*skip size, adjust size */
            NumHighBits = 1;                /* How many times to double skip size */
            while (RunHighBits != 1) {      /* While more doubling required */
                NumHighBits++;              /* Record one more doubling */
                RunHighBits >>= 1;          /* Elim one doubling from RunHighBits */
            } /* end while */
            SkipSize <=<= NumHighBits;       /* Double skip size needed num times */
            YDecSkipConst <=<= NumHighBits; /* Dble decvar diffc const to match */
            CheckShift += NumHighBits;      /* Change most sig bit rng for last run */
        } /* end if */
        YLastXStep = Y+YStep;               /* Record location new X step */
        Y += SkipSize;                       /* Increment Y by skip size */
        X += XStep;                          /* Increment X to perform X step */
        Dec += XDec2 - YDecSkipConst;        /* Update decision variable */
        XDec2 += XDec1;                      /* Update 2nd order diffc function */
        XDec1 += XDec0;                      /* Update 1st order diffc function */
        NextPlot = LinPlot;                 /* Indicate next plot type */
    } /* end if */
} /* end while */
(*LinPlot)(XEnd,YEnd,X,YLastXStep);         /* Indicate next plot type */

```

**Figure 15:** The optimized RiseRun algorithm. (X,Y) and (XEnd,YEnd) are endpoints.

If we adjust the skip size the maximum of  $|X_{End}-X|$  times, total cost is

$$(8a + 2s)|X_{End}-X| + 2a(|Y_{End}-Y| - (2^{\lfloor \log(m) \rfloor + |X_{End}-X|} - 2^{\lfloor \log(m) \rfloor})).$$

The cost of a curve segment with portions that fulfill both  $|f'(x)| < 2$  and  $|f'(x)| > 2$  will be the sum of the costs of each of the separate portions.

The closer the absolute slope of a curve is to infinity, the better the performance of the optimized RunRise algorithm. The optimized algorithm should not be used if the largest absolute slope of the plotted curve is less than 2.

## 8. 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 satisfy 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.

Because the optimized algorithms take advantage of curve shape, they require some additional care. In particular, the curve's inflection point (at which  $f''(x) = 0$ ) must also be located, resulting in the worst case in eight curve segments (again, two or three is more typical). Plotting direction must then be controlled so that the algorithms plot from the points at which  $|f'(x)| = 1$  is true, toward the points at which  $f'(x) = 0$  and  $f''(x) = 0$  are true.

## 9. AVOIDING 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 unoptimized 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  $2A_i x^3 + 2B_i x^2 + 2C_i x$ . 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

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

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  $d_1(x) = d_2(x+1) - d_2(x)$ , 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  $(X\pm 1, Y\pm 1/2)$ . 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  $(X\pm 1, f(X\pm 1))$  (see again figure 4 and table 1). Thus we have

$$d = |f(X\pm 1) - (Y\pm 1/2)| \leq 1/2.$$

Scaling by  $2D_i$  gives

$$2D_i d = |2D_i f(X\pm 1) - 2D_i(Y\pm 1/2)| \leq D_i.$$

Note now that  $2D_i f(X\pm 1) - 2D_i(Y\pm 1/2)$  is equivalent to the decision variable (4). Thus we have  $|d(x\pm 1, y\pm 1/2)| = |Dec| \leq D_i$ .

$xDec2$  represents the difference function  $d_2(x) = d(x\pm 1, y\pm 1/2) - d(x, y\pm 1/2) = g(x\pm 1) - g(x)$ , where  $g(x)$  is the function (3). We define the difference  $d_2'(x) = d_2(x)/2D_i = f(x\pm 1) - f(x)$ , where  $f(x)$  is the function (2). In the RunRise case,  $|f'(x)| < 1$ , which implies that  $d_2'(x) = |f(x\pm 1) - f(x)| \leq 1$ , and then it follows that  $|xDec2| = |d_2(x)| \leq 2D_i$ .

This allows us to conclude that representing the RunRise looping variables requires only the fulfillment of the almost trivial inequality

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

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 unoptimized RiseRun algorithm is similar to the RunRise initialization analysis. The largest intermediate value calculated during initialization is the partial sum of the decision variable  $8A_i x^3 + (\pm 12A_i + 8B_i)x^2 + (6A_i \pm 8B_i + 8C_i)x + (\pm A_i + 2B_i \pm 4C_i)$ .

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  $x_{End} = x$  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  $x_{End} \neq x$  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  $|xDec2| = d_2(x) = |8D_i f(x \pm 3/2) - 8D_i f(x \pm 1/2)| \leq 8D_i h$ , and gives us the restriction

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

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  $[0, 8D_i h]$ , and that  $|Dec| \leq 8D_i h$ . Thus the above overflow restriction for  $xDec2$  also applies to  $Dec$ .

**Table 2**

Ranges of looping variables contained in the optimized RiseRun algorithm. *bits* is the number of bits used to represent Dec, *h* is screen height in pixels

Variable	Range
RunHighBits	[0, <i>bits</i> ]
NumHighBits	[0, <i>bits</i> ]
SkipSize	[0, <i>h</i> ]
YLastXStep	[0, <i>h</i> ]
CheckShift	[0, <i>bits</i> ]
YDecSkipConst	[-XDec2,XDec2]

**Table 3**

Ranges of looping variables contained in the optimized RunRise algorithm. *bits* is the number of bits used to represent Dec, *w* is screen width in pixels.

Variable	Range
SkipSize	[0, <i>w</i> ]
XLastYStep	[- <i>w</i> /2, <i>w</i> /2]
SkipShift	[0, <i>bits</i> ]
OldXDec1	[-XDec1,XDec1]
HalfXDec1Skip	[-XDec1,XDec1]
OldXDec2	[-XDec2,XDec2]

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

$$\begin{aligned} 2^{13} D_i &< 2^{31} \\ D_i &< 2^{18}. \end{aligned}$$

Initialization in the optimized versions of the RunRise and RiseRun algorithms are practically identical to the unoptimized initializations, and thus require no new analysis. However, the looping portions of these algorithms deserve more discussion. Each of them uses the same variables used in the unoptimized algorithms. Because these variables take on the same sequence of values that they would in the unoptimized algorithms (with some values in the sequence being skipped), we can be assured that these variables will not overflow.

Both optimized algorithms change the state of several variables not used in their unoptimized counterparts. Tables 2 and 3 are tables of these variables and their ranges. RunHighBits, NumHighBits, and CheckShift are not included in table 3 because they are used in the optimized RunRise algorithm exactly as they are in the optimized RiseRun algorithm. None of the variables in these tables will overflow if we hold to the inequalities given during the analysis of the unoptimized algorithms.

The variable DecSkip in the optimized RunRise algorithm represents the difference  $d(X_{\pm(l+1)}, Y_{\pm 1/2}) - d(X_{\pm 1}, Y_{\pm 1/2})$ . In the worst case, the two decision variables in this difference will be of opposite sign, giving a worst case magnitude of  $2D_i$ . This implies that by holding to the RunRise inequalities above, we can represent DecSkip without overflow.

## 10. USE OF THESE ALGORITHMS WITH 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  $[0,1]$ . For a given value of  $t$ , the X and Y values are found, and a pixel plotted.

There are three basic problems which must be overcome if we are to render these curves with our algorithms. First, while parametric algorithms plot in the screen space  $[X,Y]$ , they calculate the individual X and Y pixel coordinates in the parametric spaces  $[t,X]$  and  $[t,Y]$ . Our algorithms, on the other hand, plot *and* calculate in screen space  $[X,Y]$ . To overcome this problem, our algorithms can be used twice: once to calculate X values, and once to calculate Y values. The algorithm variable  $x$  can be used to contain the current value of the parameter  $t$ , while the variable  $y$  contains an X or Y coordinate.

We then confront the second problem: since our algorithms can only use a step size of 1,  $x$  cannot, like  $t$ , take on values only in the range  $[0,1]$ . However, by adjusting the parametric equations, we can let  $x$  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

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

$t$  in  $[0,1]$ , the equation

$$x(t') = A(t'^3/n^3) + B(t'^2/n^2) + C(t'/n), \quad (7)$$

$t'$  in  $[0,n]$ , would describe the same coordinates.

The third problem is more subtle. As stated above, we must use our algorithms twice: once to find values for X, and once to find values for Y. As we calculate values for X, we must of course vary  $t'$  in  $[t',X]$ . But to ensure that an accurate curve is rendered, as we calculate values for Y, we must vary  $t'$  in  $[t',Y]$  in the same way. In other words,  $x(t')$  and  $y(t')$  must be evaluated at the same  $t'$  values. If both curve segments in  $[t',X]$  and  $[t',Y]$  are calculated entirely by the RunRise algorithm,  $t'$  will indeed be varied consistently: it will take on all integer values in the range  $[0,n]$ . However, if any portion of either curve segment in  $[t',X]$  or  $[t',Y]$  is calculated by the RiseRun algorithm, X or Y -- not  $t'$  -- will at some point be varied by 1, and the values  $t'$  takes on will depend on curve shape.  $x(t')$  and  $y(t')$  will be evaluated at different  $t'$  values.

We avoid this problem by ensuring that all needed curve segments in  $[t',X]$  and  $[t',Y]$  will be calculated by the RunRise algorithm alone. If no point on either of the segments has absolute slope

greater than one, they will both be plotted entirely by the RunRise algorithm. In this case, the number of parametric steps  $n$  should be greater than  $\max(|x(1)-x(0)|, |y(1)-y(0)|)$ . Otherwise, we must consider the slopes of the curves  $x(t)$  and  $y(t)$  when choosing  $n$ . The slope of (7) is

$$x(t') = (3A/n^3)t'^2 + (2B/n^2)t' + (C/n). \quad (8)$$

Clearly, as  $n$  increases, absolute slope decreases. At the same time, increasing the number of parametric steps  $n$  increases algorithm cost. We must make the minimum number of parametric steps required to ensure that only the RunRise algorithm will be used.

Slope magnitude on the two curve segments to be calculated will be greatest at one of the curves' endpoints or inflection points. At  $t = 0$ , (8) reduces to  $C/n$ . If absolute slope is to be less than one, we must have  $n \geq |C|$ . At  $t = n$ , (8) reduces to  $1/n(3A + 2B + C)$ . We must have  $n \geq |3A + 2B + C|$ . The inflection point lies at  $-B/3A$ . If  $0 < -B/3A < 1$ , we must also have  $n \geq |-B^2/3A + C|$ . In summary, the number of parametric steps  $n$  which must be made is the largest of the following values:

$$\begin{aligned} & \lceil |C_x| \rceil; \lceil |C_y| \rceil \\ & \lceil |3A_x + 2B_x + C_x| \rceil; \lceil |3A_y + 2B_y + C_y| \rceil \\ & \lceil |-B_x^2/3A_x + C_x| \rceil; \lceil |-B_y^2/3A_y + C_y| \rceil \\ & \lceil |x(1)-x(0)| \rceil; \lceil |y(1)-y(0)| \rceil \end{aligned}$$

where  $A_x$ ,  $B_x$ , and  $C_x$  are the coefficients of  $x(t)$ , and  $A_y$ ,  $B_y$ , and  $C_y$  are the coefficients of  $y(t)$ .

By choosing  $n$  as indicated, we are finding the largest  $t$  increment which will ensure that  $X$  or  $Y$  will never be incremented by more than one, and using it over both of the curve segments  $x(t)$  and  $y(t)$ . As a result, we oversample in those portions of the curve which would allow larger  $t$  increments. However, since these portions of the curve will have slope of low magnitude, use of the optimized RunRise algorithm should compensate for most of this additional cost. A different approach is taken by algorithms that use adaptive forward differencing, which uses some additional looping operations to dynamically adjust the size of  $t$  increments to ensure that  $X$  and  $Y$  steps are always close to one.

## 11. AVOIDING OVERFLOW WITH PARAMETRIC CURVES

The parametric version of the RunRise decision variable (4) is

$$A_i(t \pm 1)^3 + B_i n(t \pm 1)^2 + C_i n^2(t \pm 1) - D_i n^3(X \pm 1/2).$$

Note that we have not scaled the parametric decision variable by 2 because we know that  $n$  is an even number. Below, we present overflow analysis only for the parametric equation  $x(t)$ . For overflow restrictions for  $y(t)$ , simply substitute  $h$  for  $w$ . The RunRise parametric differences are:

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

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

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

For the parametric versions of our algorithms, we only present overflow restrictions for the looping sections. Analysis for the parametric RunRise variables  $x_{Dec2}$  and  $Dec$  is quite similar to the analysis for the identically named non-parametric RunRise variables, and gives the overflow restriction

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

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

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  $n \leq 1290$ .

## 12. ALGORITHM COMPARISON AND EVALUATION

In this section, we compare the RunRise algorithms as they would be used with parametric curves with the algorithms A and B presented by Klassen in [11]. We do not take into account the cost of using Klassen's algorithms with adaptive forward differencing, nor the cost associated with RunRise oversampling. Table 4 shows the operation costs and the overflow restrictions associated

**Table 4**

A comparison of Klassen's algorithms from [11] with our algorithms 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.

Algorithm	Operation Cost											Overflow Restriction
	Main Loop				Initialization							
	+	if	<<	=	+	*	div	xdiv	<<	if	=	
RunRise	4.1	2	0	4	16	5	0	0	11	2	16	$n^3 < 2^{bits-1}$
RunRiseOpt	2.2	0.9	0.4	2.2	16	5	0	0	13	4	22	$n^3 < 2^{bits-1}$
Klassen's A	4	1	3	4	$17+3Z$	12	1	5	2	$Z$	$28+2Z$	$46wn < 2^{bits-1}$
Klassen's B	11	4	0	11	40	12	5	0	0	6	33	$2n^3 < 2^{bits-1}$

with Klassen's algorithms and our algorithms. We have assumed that a barrel shifter is available, and counted all looping operations. For initialization, we follow Klassen's practice of weighing each branch of a conditional statement equally.

We averaged the performance of the main loops of our algorithms over 100 curves of the form  $y = Ax^3$ , with  $A$  varying between  $1/50,000$  and  $1/500$ . The curve segments were chosen so that skipping was performed over the entire segment. Testing showed that with other types of segments, optimized algorithm performance was only comparable to unoptimized algorithm performance.

Klassen's algorithm A has no conditional statements and thus is trivially averaged. Algorithm B, however, is not averaged and is shown as presented by Klassen.

Clearly, the main loop of the unoptimized algorithm uses less operations than the loop in Klassen's algorithm B, and is comparable to the loop in algorithm A. The main loop of the optimized algorithm clearly outperforms the loops in both of Klassen's algorithms. 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 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 algorithms and Klassen's algorithm B both have restrictions that are cubic in  $n$ , with the RunRise algorithms allowing twice as many parametric steps as algorithm B.

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 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 (`xdiv`) 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.

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 and some extra overhead, 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.

## 12. CONCLUSIONS AND FUTURE WORK

We have presented integer-only algorithms that allow fast, accurate plotting of cubic curves. We have also presented optimized algorithms that work even more quickly when curve slope nears infinity or zero. 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.

### 13. 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.