A VLSI Algorithm for Integer Forward Differencing

Hannes Hassler

Department of Information Engineering
Nagoya University

Abstract. We propose a new initialization procedure for Forward Differencing. Forward Differencing has applications in computer graphics and the control of Computer Numerical Control machines, allowing a quick generation of nonlinear curves. In this article we develop a procedure to set up FD. We analyze the error caused by this initialization and compare it to a conventional method. In the course of the analysis we derive practical parameters which match standard graphical resolution.

1 Introduction

Computer graphics and Computer Numerical Control machine tools require frequently a scan-conversion of nonlinear curves. Scan-conversion algorithms compute the coordinates of the pixels that lie on or near an ideal, infinitely thin curve imposed on a 2D raster grid. Many algorithms have been proposed for a fast generation of straight and curved lines. All these methods generate plotting movements or pixel coordinates with a few fast operations such as addition and comparison and try to avoid multiplication, if possible [1, 9]. Moreover, several works have also replaced floating point numbers by the faster integer arithmetic [1, 3, 5].

One basic method to generate curves is Forward Differencing (FD). Works on curve generation by FD date back at least to the 60's [2, 3, 5, 11]. In this article we propose a new initialization to set up FD and compare it to the traditional one. (Other termini for FD are *tabulation* [6] or *DDA* (for Digital Differential Analyzer) [8, 11].) Usually, FD is programmed using floating point numbers, which are rounded to integers at each pixel. Recent works by Field and Klassen [3, 5] consider integer FD algorithms instead of the slower floating point FD. This article employs integer FD, which is easier to implement in hardware. The main problems with integer FD is the possible overflow and the loss of precision.

Our method uses implicitly a splining polynomial of degree m between an equidistant point sample $P = \{P_0, \cdots, P_m\}$ where P_0 and P_m mark the beginning and endpoints of the approximated curve, respectively. We assume that the point sample P can be generated with low costs compared to the generation of the curve itself. The splining process to set up FD consists of a simple matrix-vector multiplication. The matrix is a preprocessed constant and the vector is simply the point sample P.

In Section 2 we explain FD, the function restrictions and basic formulas. The traditional and a new initialization method are presented in Section 3. Section 4 proposes the FD algorithm which uses this initialization. Finally, in Section 5 we analyze the different kinds of numerical errors which arise during initialization and in the course of algorithm.

2 Forward Differencing

FD evaluates a function $f(x)$ at $n+1$ equidistant points $x_0, \cdots, x_n$ as for instance, in a plotting task. We assume, which is customary in computer graphics, that the lower left corner of our graphical device refers to the point $(0,0)$ and the upper right corner refers to $(1,1)$ [Fig.1]. The horizontal distance of two consecutive points is thus determined by the resolution $r \times r$, and it is $\Delta = x_i - x_{i-1} = 1/(r-1)$.

We assume smooth functions $f(x)$ for which its derivatives $f^{[k]}(x)$ for $1 \leq k \leq m$ exist and which pass the condition

$$0 \leq f(x) \leq 1, \;\; |f^{[k]}(x)| < 1 \quad \text{for } \; 0 \leq x \leq 1 \tag{1}$$

This condition is neccessary to avoid overflow of the values R_i of the algorithm in Section 4. Practical f's may not satisfy the derivative condition, but we can adapt functions by applying following substitution:

Assume that $f(x)$ does not meet the restrictions in the intervall $[0,1]$. We substitute $f(x)$ by the two functions $g_1(x) := f(x/2)$, $g_2(x) := f((x+1)/2)$ and tabulate each of them in the interval $[0,1]$. Note that $g_1^{[j]}(x) = 2^{-j} f^{[j]}(x/2)$ and the analogy accounts for $g_2(x)$. One must be well aware, though, that the step size is now only half a pixel. If one (or both) of these $g(x)$'s has still too large derivatives it can be split up further. Of course, this substitution requires more iterations and also more initialization phases.

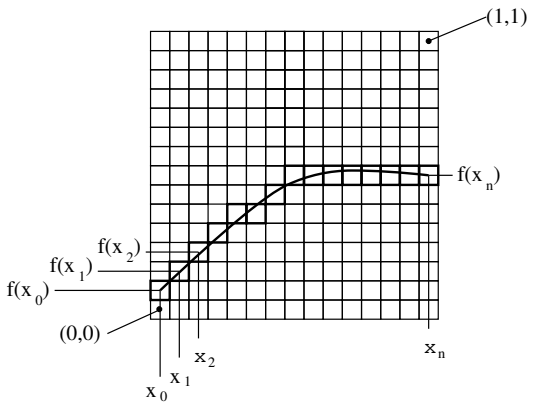

Figure 1: *Graphical display of a continuous curve*

FD generates the consecutive function values $f(x_0), f(x_1), f(x_2), \cdots, f(x_n)$ using the tabulation vector $\mathbf{F}(i) := (F_0(i), F_1(i), \cdots, F_m(i))^T$ and the recursive formula:

$$F_j(i+1) := F_j(i) + F_{j+1}(i) \quad \text{for} \quad j = 0, \cdots, m-1 \;\; (\text{in this order}) \tag{2}$$

Each iteration adds to the *derivative* $F_j(i)$ the nexthigher *derivative* $F_{j+1}(i)$ To set up FD one has to calculate according initial values $F_0(0), F_1(0), \cdots, F_m(0)$. With the correct set up the first vector component $F_0(i)$ assumes consecutively for each recursion iteration the values $f(x_i)$. (The IPT set up starts with $m+1, m+2, \cdots$, the TASP set up starts with $0, 1, \cdots$. In Section 3 we explain these two methods.) Condition (1) and formula (2) allow to prove two basic properties (3), (4) of the F_j's. The first property we prove by the following generalization of the mean value theorem (proven in the appendix).

Theorem 1 *For all $F_j(i)$ there exists a ξ_{ji} such that*

$$\exists \xi_{ji} \in (x_i, x_{i+j}) \qquad F_j(i) = f^{[j]}(\xi_{ji})\Delta^j$$

We apply condition (1) and get

$$|F_j(i)| < \Delta^j = (r-1)^{-j} \tag{3}$$

which is an upper limit for each $|F_j(i)|$, respectively. In other words, the fix point values of each $F_j(i)$ has right of the radix point at least $j \log_2 \Delta$ bits which are zero. This property motivates a *stair shift* design for the FD circuit [Fig.2]. Successive addition stages include a shift by $-\log_2 \Delta$ bits. Such a stair shift design allows a higher absolute accuracy for higher $F_j(i)$'s. In Section 5 we show that such a design using l-bit registers limits the maximum truncation error of $F_0(i)$, its absolute values being lower than $2^{-l}r$.

The second property is the Gregory-Newton formula [6, 7], shown easily by induction.

$$F_0(i) = \sum_{j=0}^{m} F_j(0) \binom{i}{j} \tag{4}$$

This formula allows to calculate the $F_0(i)$ values directly from the initial values. Otherwise, FD generates them by the recursion (2). The Gregory-Newton formula is essential to the splining process and the error estimation.

3 Tabulation Initialization

3.1 Initial Points Tabulation (IPT)

The usual method [6] determines a starting vector $\bar{\mathbf{F}}(0)$ by choosing the first $m+1$ values of the $(n+1)$–sequence $f(x_0) \cdots f(x_n)$ as the sample vector. Hence, we call it Initial Points Tabulation, IPT for short, and denote its sample vector with $\bar{\mathbf{S}}$. It works as follows:

For $i = 0$ To m Do $\bar{F}_i(m) := f(x_i)$
For $i = 1$ To m Do
{(I1) For $j = i$ To m Do $h_j := \bar{F}_j(m) - \bar{F}_{j-1}(m)$
(I2) For $j = i$ To m Do $\bar{F}_j(m) := h_j$ }

Because this set up generates approximation values starting with m we denote the initial vector by $\bar{\mathbf{F}}(m)$ which has not the initial parameter 0 as the other set up method. After setting up $\bar{\mathbf{F}}(m)$ the recursion (2) generates the approximation values with $F_0(m+1) = \bar{f}(x_{m+1}), \cdots, F_0(n) = \bar{f}(x_n)$.

3.2 Tabulation by Splining (TASP)

In this article we propose an alternative set up method which we call *Tabulation by Splining*. TASP is based on using reversely the Gregory-Newton formula (4). From a sample of equidistant function values we calculate the initial vector. To distinguish this vector from the IPT one we denote it by $\tilde{\mathbf{F}}(0)$ and the sample vector by $\tilde{\mathbf{S}} = (f(x_0), f(x_d), \cdots, f(x_{md}))^T$, accordingly. (we assume $d = n/m$) With the Gregory-Newton formula we set up the system with the binomial factors

$$\underbrace{\begin{pmatrix} \binom{0}{0} & \binom{0}{1} & \cdots & \binom{0}{m} \\ \binom{d}{0} & \binom{d}{1} & \cdots & \binom{d}{m} \\ \vdots & \vdots & \ddots & \vdots \\ \binom{md}{0} & \binom{md}{1} & \cdots & \binom{md}{m} \end{pmatrix}}_{V_{d,m}} \underbrace{\begin{pmatrix} F_0(0) \\ F_1(0) \\ \vdots \\ F_m(0) \end{pmatrix}}_{\tilde{\mathbf{F}}(0)} = \underbrace{\begin{pmatrix} f(x_0) \\ f(x_d) \\ \vdots \\ f(x_{md}) \end{pmatrix}}_{\tilde{\mathbf{S}}} \tag{5}$$

The matrix $V_{d,m}$ is essentially a Vandermonde matrix. We get the sought starting vector $\tilde{\mathbf{F}}(0)$ by the matrix-vector multiplication

$$\tilde{\mathbf{F}}(0) = V_{d,m}^{-1} \tilde{\mathbf{S}} \tag{6}$$

$\tilde{\mathbf{F}}(0)$ is equivalent to the polynomial received by interpolation between the sample points $\tilde{\mathbf{S}}$. In practical applications one or (if needed) several such matrices $V_{d,m}^{-1}$ can be precomputed. The degree of the (implicit) polynomial is equal to m. The calculation of this matrix $V_{d,m}^{-1}$ is not considered being part of the online computation and can be done with an arbitrary high precision.

3.3 Comparison of IPT with TASP

IPT has the advantage that it needs only $m(m+1)/2$ subtractions. In comparison, TASP needs $(m+1)^2$ multiplications and $m(m+1)$ additions.

One advantage of TASP is its lower initial error. In Section 5 we find that the initial vector $\tilde{\mathbf{F}}(0)$ of TASP is generated with much lower an error than $\bar{\mathbf{F}}(0)$ of IPT. This difference allows a higher polynomial degree m for TASP than for IPT.

Another advantage of the TASP polynomial is its overall lower error. If the function which we want to approximate is a polynomial of a degree equal to or smaller than m both IPT and TASP generate the unique starting vector $\mathbf{F}(0)$. However, this is generally not the case. Usually, IPT and TASP can only approximate the given function, with TASP having the better error behavior. The IPT-polynomial approximates the function better only during the initial iterations, but degrades quickly [Fig.3]

Another question which is difficult to answer is the condition under which TASP tabulation is faster than the explicit calculation of the function values. This depends on the costs for calculation of the function, its maximum derivatives, the number of values to be generated and on the speed of floating point operations versus integer additions.

The costs of TASP (without splitting) are per pixel m integer additions and about m^2/n floating point multiplications and additions. However, if the function has to be split once, these costs increase by the factor 2, at least. Another split drives this factor to 4 and so on.

On the other hand, a direct function evaluation (of an m-degree polynomial) needs per pixel m floating point additions and $m+1$ floating point multiplications. Even if we have to split the function only twice the direct evaluation might be already faster.

4 The FD algorithm

One reason for using integers is to make the algorithm attractive to implement it in VLSI, too. For such a hardware algorithm we make a few assumptions and propose the following model.

1. The circuit consists of $m+1$ registers $R_0 \cdots R_m$ and a counter C. Each register R_i stores l bits and has an overflow detection (including another auxiliary bit).

2. The primitive 'CheckOutput(C,R)' checks the values C and R if they are valid integers for the range of the graphical device. If they are valid they are output to the graphical device. Otherwise an interrupt is generated.

3. A coordinate for the graphical device consists of a Z-bit integer pair. It does not matter if the graphical device receives two or more identical coordinates.

The notation $\gg Z$ means shift right by Z bits. Initially, FD is called with the parameters FD$(f, 0, 1, 1)$. The circuit model is sketched in Fig.2.

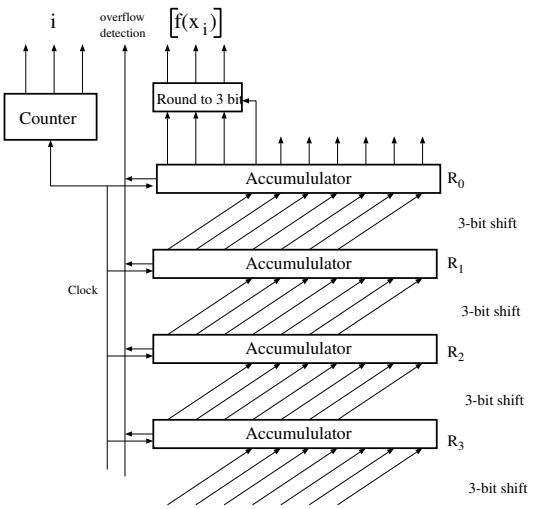

Figure 2: *Schematic circuit for step (T) of the Tabulation Algorithm with $Z = 3$ and $l = 10$. It generates successively the rounded value $[f(x_i)]$ and i, accordingly.*

4.1 The Algorithm

Tabulation Algorithm

(V) Calculate $V_{m,d}^{-1}$ (offline)

FD(f, a, b, D)

(I) Generate $\tilde{\mathbf{F}}(0) := V_{m,d}^{-1}(f(t_0) \cdots f(t_m))^T$ with $t_j = a + \frac{b-a}{m} j$.

(H) If $Error - heuristic(\tilde{\mathbf{F}}(0)) > 2^{-Z+\text{const}}$
Then (G); exit procedure FD(f, a, b, D)

(L) $R_j := \tilde{F}_j(0)$; $C := a$

(T) For $i = 0$ To md Do
(T1) CheckOutput(C,$(R_0 + 2^{l-Z-1}) \gg (l - Z)$)
(T2) $C := C + D$
(T3) For $j = 0$ To $m - 1$ Do
(T3.1) $R_i := R_i + (R_{j+1} \gg Z)$
(T3.2) If $R_i \geq 1$ Then (G); exit loop (T)

(G) FD$(f, a, (a + b)/2, D/2)$
FD$(f, (a + b)/2, b, D/2)$

In this algorithm the registers R_j $(0 \leq j < m)$ are constantly updated. The register R_0 contains the current function value $0 \leq f(x_i) \leq 1$. For output the most significant $Z+1$ bits of R_0 are rounded into a Z bit integer. Fig. 2 sketches a circuit neccessary for the parallel execution of the loop (T) of the algorithm. In the next section we study the error behavior of TASP and compare it with that of IPT.

4.2 A non-equidistant example

Up to now we have considered equidistant tabulation, i.e. $x_i - x_{i-1}$ is constant. This is not always the case. However, if we can assume that both coordinates are generated by smooth parametric functions $y(t), x(t)$ we can extend the splining to the x-coordinates as well.

Consider following example for a CAD application. Assume we store 3-dimensional (smooth) lines and have to generate quickly 2-dimensional projections. The 3D line is stored by a sample of 3-dimensional points $\{x(t_0), y(t_0), z(t_0)\}, \cdots \{x(t_m), y(t_m), z(t_m)$. The plane onto which we are going to project the line is described by two vectors $\vec{v}_x$ and $\vec{v}_y$. We multiply each 3D point with both vectors and get the projection $\{x'(t_0), y'(t_0)\}, \cdots, \{x'(t_m), y'(t_m)\}$. Neither the x' nor the y' are neccessarily equidistant. Next, we multiply both the x' and y' point sample by the matrix $V_{d,m}^{-1}$. We get two starting vectors for generating coordinates, recursively. One recursion generates the $x'(t_i)$'s, the other $y'(t_i$'s.

5 Error Behavior

Integer TASP approximation has to deal with four different kinds of error. The first one (oscillation) is the principal error caused by approximating an infinite polynomial by a finite one. We estimate this error online by a heuristic. The second one (initialization error) results from the limited precision (preferrably double precision) when $\tilde{\mathbf{F}}(0)$ is calculated. For this error we can derive offline upper bounds. The third type of error is accumulated by the repeated recursion (2) and caused by the truncation in step (T2.1). For this error we can derive offline upper bounds, too. The fourth error is a possible overflow and hardware detected. If it occurs an interrupt is generated and the functions substituted as explained in Section 2.

5.1 Oscillation error

Both IPT and TASP approximate a given function $f(x)$ by a polynomial of degree m. The number of components of the vector $\mathbf{F}(0)$ are accordingly $m+1$. Typically, an equidistant polynomial approximation equals the function only at $m+1$ points and oscillates up and down otherwise As a rule of thumb the TASP-error is roughly at its maximum in the middle of any two adjacent sample points. We use this heuristic error estimation and calculate the maximal error in the middle of the first and second sample point with formula

$$\textit{maximal TASP} - \textit{error} \approx \left| f(\lfloor d/2 \rfloor / n) - \sum_{j=0}^{m} \tilde{F}_j(0) \binom{\lfloor d/2 \rfloor}{j} \right| \tag{7}$$

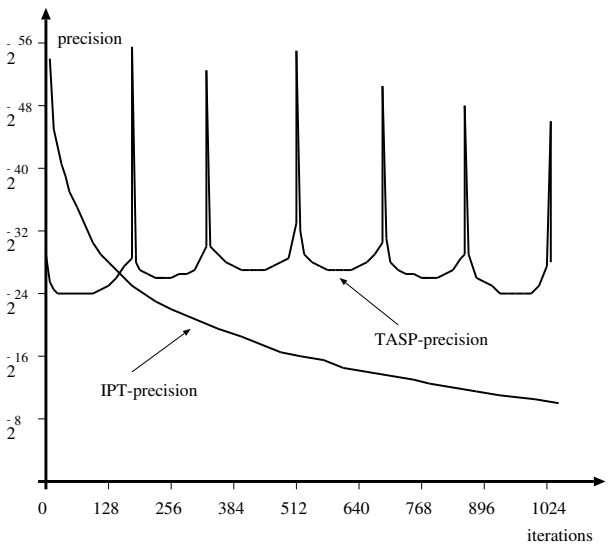

Figure 3: *Precision diagram due to TASP and IPT initialization, respectively. The approximated function is* $\sin(x)$ *in the interval* $[0,1]$ *and the degree of the respective polynomials is* $m = 6$.

In contrast, the IPT point sample is taken at the very beginning of the approximation interval. Typically, the approximation deteriorates to its maximum error at the end

$[a,b]$	m	d	$f(x)$	TASP-error (PEE)	IPT-error (PEE)
$[0,1]$	3	341	$\sin(x)$	$2^{-11.8}(\max\{2^{-12.3}, 2^{-22}\})$	$2^{-6.9}(2^{-6.9})$
$[0,1]$	4	256	$\sin(x)$	$2^{-15.2}(\max\{2^{-15.3}, 2^{-22}\})$	$2^{-7}(2^{-7})$
$[0,1]$	5	205	$\sin(x)$	$2^{-20}(\max\{2^{-20.8}, 2^{-22}\})$	$2^{-12.3}(2^{-12.3})$
$[0,1]$	6	171	$\sin(x)$	$2^{-22.5}(\max\{2^{-24.2}, 2^{-22}\})$	$2^{-10.3}(2^{-10.3})$
$[0,1]$	7	146	$\sin(x)$	$2^{-22.9}(\max\{2^{-29.9}, 2^{-22}\})$	$2^{-2.3}(2^{-2.3})$
$[2,3]$	3	341	$\ln(x) - 1/2$	$2^{-13.4}(\max\{2^{-13.5}, 2^{-22}\})$	$2^{-6.4}(2^{-6.4})$
$[2,3]$	4	256	$\ln(x) - 1/2$	$2^{-16.8}(\max\{2^{-17}, 2^{-22}\})$	$2^{-7.8}(2^{-7.8})$
$[2,3]$	5	205	$\ln(x) - 1/2$	$2^{-20.1}(\max\{2^{-20.3}, 2^{-22}\})$	$2^{-7.2}(2^{-6.8})$
$[2,3]$	6	171	$\ln(x) - 1/2$	$2^{-23.2}(\max\{2^{-23.5}, 2^{-22}\})$	$2^{-2.6}(2^{-2.6})$
$[2,3]$	7	146	$\ln(x) - 1/2$	$2^{-23.3}(\max\{2^{-26.7}, 2^{-22}\})$	$2^{9.6}(2^{9.6})$

Table 1: *TASP's and IPT's maximum errors in the respective interval* $[a,b]$. *The initial values were computed with double precision (rounding to 56 bits), iterations with* $l = 32$*-bit truncation. In brackets are the heuristically* pre-estimated errors *(PEE) where TASP uses formulas (8) and (15), IPT uses (9). The product* md *roughly equals 1024 (generated pixels).*

of the interval and hence, we estimate its error by

$$\textit{maximal IPT} - \textit{error} \approx \left| f(1) - \sum_{j=0}^{m} \bar{F}_j(0) \binom{n}{j} \right| \tag{8}$$

Fig.3 diagrams the specific error behavior of IPT and TASP, both approximating $\sin(x)$ with a 6-degree polynomial. The precision function of TASP depicts clearly its sample points at i being a multiple of $d = 171$ (the peaks denote very small errors). Note that in this figure we computed both initialization and recursion with double precision (56 bit rounding) to avoid the truncation errors and to illustrate the principal precision of either method.

In comparison, the data in Table 1 has been generated assuming double precision (56 bit) for the initialization and 32 bit precision for the recursion. In a practical application the recursion iteration would be done with integers which have a lower precision than the number format used for initialization. Note, that these integers represent the fix point fraction numbers $F_j(i)$.

5.2 Initialization Error

The absolute error vector is the difference $|\tilde{\mathbf{F}} - \mathbf{F}|$. Due to property (3) we introduce a special kind of relative error. We define the *effective error vector* $\mathbf{E}(i)$ by

$$E_j(i) := |\tilde{F}_j(i) - F_j(i)|/\Delta^j \tag{9}$$

The effective error vector tells how many of the most significant nonzero bits of $\tilde{\mathbf{F}}(0)$ are correct. For instance, consider the initial effective error vector of the IPT-approximation in Fig.3. We calculated it with a precision of 56 bits and monitored the error with a high precision package. We received initial effective errors

$\bar{\mathbf{E}}(0) = (0, 0, 2^{-51}, 2^{-36}, 2^{-23}, 2^{-12}, 2^{-0.5})^T$. Note that the large effective error of the last component renders $\bar{F}_6(0)$ as useless. See also Table 1 where the according IPT polynomials of degree 5 have better precision than those with degree 6.

In contrast, the initial effective errors generated by the matrix multiplication in TASP are much smaller. In the special example of Fig.3 we found them to be $\tilde{\mathbf{E}}(0) = (0, 2^{-54}, 2^{-48}, 2^{-44}, 2^{-41}, 2^{-40}, 2^{-37.7})^T$. Following theorem can be applied to derive upper bounds on rounding errors in advance.

Theorem 2 *The rounding error ε in calculating the sum $u_1 + \cdots + u_k$ as s-bit precision floating point numbers has an upper limit*

$$\varepsilon < ((1 + 2^{-s})^k - 1) \sum_{j=1}^{k} |u_j|$$

The proof can be found in the appendix. We use this theorem to derive an offline upper bound on the TASP initialization error. Note that the components of the sample vector $\tilde{\mathbf{S}}$ in $V_{d,m}^{-1}\tilde{\mathbf{S}}$ are $0 \leq S_i \leq 1$. A vector $\mathbf{U}$ of upper bounds of the effective initial roundig error of $\tilde{\mathbf{F}}$ is thus

$$U_i = ((1 - 2^{-s})^{m+1} - 1)\Delta^{-i} \sum_{j=0}^{m} |V_{d,m}^{-1}\{i, j\}| \tag{10}$$

and $\tilde{\mathbf{E}}(0) < \mathbf{U}$. For the above example we get the upper bounds $\mathbf{U} = (2^{-53}, 2^{-46}, 2^{-41}, 2^{-37}, 2^{-35}, 2^{-33}, 2^{-32})^T$.

5.3 Truncation Error

During the recursive iteration the registers R_j accumulate growing truncation errors. To denote the upper bound for the accumulated error of register R_j at step i we use $T_j(i)$. Following theorem presents an upper bound.

Theorem 3

$$T_0(i) < 2^{-l} r$$

We see the error accumulation depends on the number of iterations, but is independent from the degree of the polynomial, compare Tab.1. In contrast, related work of Klassen [5] citing Shantz and Chang [10], which can not rely on condition (19), (they estimates the error essentially with an initial $T_j(0) = 2^{-l-1}$) suggest polynomials with a degree higher than 3 not being feasible.

5.4 Overflow detection

To enable fast addition we propose redundant binary adders [12]. The value of an l digit redundant binary number (with an $(l+1)$th auxiliary digit) $[b_l \cdots b_0]$ is $\sum_{s=0}^{l} b_s 2^s$ with $b_s \in \{-1, 0, 1\}$.

However, redundant binary numbers make an exact overflow detection very complicated and time consuming. Hence, we propose a *relaxed* overflow detection instead.

To decide if an overflow has happened the three most significant digits are evaluated. An overflow interrupt is generated if for any of the registers R_j there is

$$\left| \sum_{s=l-2}^{l} b_s 2^{s-l} \right| \geq 1.25 \tag{11}$$

Such an overflow detection serves our purposes just as well, because there occurs no overflow which is not either contained in the auxiliary bit or causes an overflow interrupt. Note that in the case of an overflow interrupt the actual value of the according fractional number may be smaller than 1.25, but is certainly larger than 1.

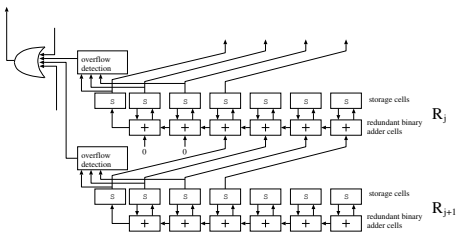

Figure 4: *Relaxed overflow detection*

6 Conclusion

We have presented a new initialization method for Forward Differencing. This integer FD algorithm is attractive for implementation in VLSI, too. The algorithm tabulates smooth spline functions, generating pixel coordinates (or plotter movements) at a speed of one pixel/iteration. In principle, it works with equidistant sample values. However, it is possible to assume that both the x and the y coordinate rely on an implicit and equistant parameter. We have sketched one possible application.

Acknowledgement

This work was supported by Monbusho scholarship. I thank Professor Naofumi Takagi from Nagoya University for many helpful discussions. Among other things he encouraged me to consider a relaxed overflow detection. I also thank Donald E. Knuth from Stanford University for his critical and friendly letter regarding the proposed method.

Appendix

Proof of Theorem 1. We define recursively for $0 \leq j \leq m$ the functions $g_0(x) := f(x)$ and (with the constant Δ)

$$g_j(x) := \frac{g_{j-1}(x+\Delta) - g_{j-1}(x)}{\Delta} \tag{12}$$

Since $F_0(i) = f(x_i) = g_0(x_i)$ we can derive recursively, by formula (2)

$$g_j(x_i) = F_j(i)/\Delta^j \tag{13}$$

By the mean value theorem we know that there exist for every j (and x) a $\xi_j \in (x, x+\Delta)$ such that $g_j(x) = \frac{g_{j-1}(x+\Delta)-g_{j-1}(x)}{\Delta} = g'_{j-1}(\xi_j)$ Applying the mean value theorem again shows that there exists a $\xi_{j-1} \in (\xi_j, \xi_j + \Delta)$ $g'_{j-1}(\xi_j) = \frac{g'_{j-2}(\xi_j+\Delta)-g'_{j-2}(\xi_j)}{\Delta} = g''_{j-2}(\xi_{j-1})$ By successive application of the mean value theorem we show finally that there exists a $\xi_1 \in (x, x+j\Delta)$ such that

$$g_j(x) = g_0^{[j]}(\xi_1) = f^{[j]}(\xi_1) \tag{14}$$

Especially, for every x_i there is a $\xi_{ji} \in (x_i, x_{i+j})$ such that [using equation (13)]

$$f^{[j]}(\xi_{ji}) = g_j(x_i) = F_j(i)/\Delta^j \tag{15}$$

which shows the theorem.

Proof of Theorem 2. We denote the absolute rounding error of u with $R(u)$. It is easy to see that $R(u) < |u|/2^s$ and

$$R(t+u) \leq R(|t| + |u|) \leq (|t| + |u|)/2^s \tag{16}$$

In the worst case rounding errors add up. We can estimate the rounding error from above by considering the addition of the absolute values $|u_i|$. Without loss of generality we assume that there is an always positive rounding error and we get the successive upper bounds L_i of the partial sums including the rounding errors. Obviously, there is $L_1 = |u_1|(1 + 2^{-s})$ and

$$L_i = (L_{i-1} + |u_i|)(1 + 2^{-s}) \quad 2 \leq i \leq k \tag{17}$$

With induction we prove

$$L_i \leq \sum_{j=1}^{i} |u_j|(1 + 2^{-s})^i \tag{18}$$

For $i = 1$ it is true. Let us assume that the inequality is proven up to $i - 1$. It is then

$$L_i \leq (\sum_{j=1}^{i-1} |u_j|(1 + 2^{-s})^{i-1} + |u_i|)(1 + 2^{-s}) < (\sum_{j=1}^{i} |u_j|(1 + 2^{-s})^{i-1})(1 + 2^{-s})$$

which is

$$L_i < \sum_{j=1}^{i} |u_j|(1 + 2^{-s})^i$$

The theorem follows from $\epsilon < L_k - \sum_{j=1}^{k} |u_j|$.

The least significant bit at register R_j has the weight $2^{-l}\Delta^j$. This weight is an upper bound for the error happening during step i at register R_j. To denote the upper bound for the accumulated error of register R_j at step i we use $T_j(i)$. If we calculate the initial vector $\mathbf{F}(0)$ with TASP we can use formula (10) to guarantee a certain initial rounding error no bigger than a certain limit. Hence, we know that the initial error is less than $T_j(0)$ (for $j = 0 \cdots m$) and

$$T_j(0) = 2^{-l-1}\Delta^j \tag{19}$$

(In the above example we have $l = 32$ and **U** guarantees us that the effective error is at most 2^{-32}.) The recursive addition allows to develop following recursive formula for the upper error bounds

$$T_j(i) = T_j(i-1) + 2^{-l-1}\Delta^j + T_{j+1}(i-1) \quad \text{for } 0 \le j < m \tag{20}$$

It is easy to prove that

$$T_j(i) = \sum_{k=1}^{m-j} 2^{-l-1}\Delta^{j+k-1} \binom{i+1}{k} \tag{21}$$

which fulfills the initial condition (19) and the recursive equation (20). For the upper bound on the error of the final result we estimate $E_0(i)$. Since $i \le 1/\Delta$ we have the upper bound

$$T_0(i) < \sum_{j=1}^{m} 2^{-l-1}\frac{i+1}{j!} < 2^{-l}(i+1) \le 2^{-l}r \tag{22}$$

References

[1] J. Bresenham, *A Linear Algorithm for Incremental Digital Display of Circular Arcs*, CACM 20, No. 2, p.100-106, Feb. (1977)

[2] P. E. Danielsson, *Incremental Curve Generation*, IEEE Trans. Comput. C-19, No. 9, p.783–793 Sept. (1970)

[3] D. Field, *Incremental Linear Interpolation*, ACM Trans. Graph. 4, p.1–11, Jan. (1985)

[4] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, *Computer Graphics*, Reading, MA Addison-Wesley, 2nd ed. (1990)

[5] R. V. Klassen, *Integer Forward Differencing of Cubic Polynomials: Analysis and Algorithms*, ACM Trans. Graph. 10, No. 2, p.152–181, April (1991)

[6] D.E. Knuth, *The Art of Computer Programming*, Vol 2, Reading, MA: Addison Wesley, 2nd ed. p.469 (1981)

[7] M.C. Kohn, *Practical Numerical Methods: Algorithms and Programs*, Macmillian Publishing Company, New York, p.127 (1987)

[8] W.M. Newman, and R.F. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York (1979)

[9] M.L.V. Pitteway, *Algorithm for drawing ellipses or hyperbolae with a digital plotter*, Comptr. J. 10,3, p.282–289 Nov. (1967)

[10] M. Shantz and S-L. Chang, *Rendering trimmed NURBS with adaptive forward differencing*, Comput. Graph. 22,4, p. 189–198, Aug. (1988)

[11] T.H.R. Sizer, *The Digital Differential Analyser*, Chapman and Hall Ltd., London (1968)

[12] N. Takagi, H. Yasuura, S. Yajima, *High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree*, IEEE Trans. Comput. v. c-34, No 9, p.789–796 Sept. (1985)