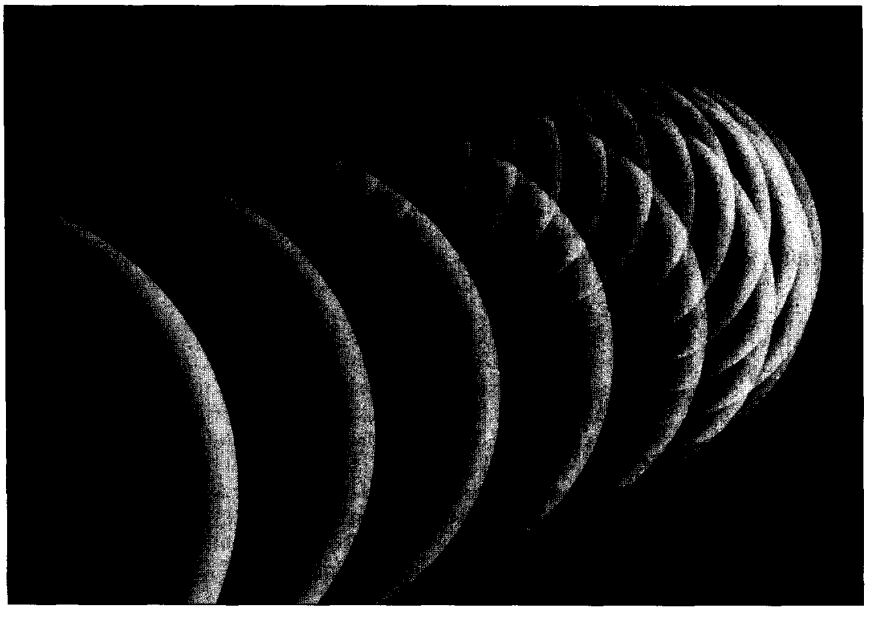
Double- and Triple-Step Incremental Linear Interpolation



Phil Graham and S. Sitharama Iyengar
Louisiana State University

This variable-step algorithm can reduce the double-step algorithm's loop iterations by 12.5 percent on average, while keeping the same worst-case performance, code complexity, and initialization costs.

Incremental linear interpolation determines the set of $n + 1$ equidistant points on an interval $[a, b]$ where all variables involved (n, a, b, and the set of equidistant points) are integers and $n > 0$. The interpolated points are denoted as x_i, $0 \le i \le n$ and defined by rounding off the following mathematical expression:

$$x_i = a + [(b - a)/n]i = a + ki \quad \textbf{(1)}$$

where $k = (b - a)/n$. Often, interpolation algorithms must also be reversible—that is, the points produced when interpolating from a to b must be the same as those produced when interpolating from b to a.

Compared with integer addition, multiplication is a very time-consuming operation. Therefore, algorithms that involve interpolation, such as the simulation of lighting effects and other computer graphics and numerical applications, can be very slow if they use multiplication to perform interpolation in a straightforward manner. As noted in earlier work, the problem of digitizing a line segment is quite similar to linear interpolation. Attempts to increase interpolation speed have therefore focused on using techniques developed for line-drawing algorithms to reduce the logic and number of multiplications required.

Field[1] developed a fixed-point variant of the digital differential analyzer[2] and called the variant *A3*. He also presented a generalization of Bresenham's line-drawing algorithm,[3] which he called *B5*. Both these algorithms are of the single-step type, since each iteration produces only one new point. However, A3 has two disadvantages: Its speed depends on the presence of a barrel shifter and it can lose accuracy. B5 needs no additional hardware and has no error.

Rokne and Rao[4] used an approach based on the *double-step line-drawing algorithm*[5] to perform linear interpolation. In the resulting algorithm, each iteration determines two points while using basically the same amount of logic as B5. Consequently, their *double-step interpolation algorithm* can be roughly twice as fast as B5. While interpolation algorithms based on other line-drawing algorithms with larger fixed-step sizes[6] might be faster, they also become much more complex.

We have recently shown, however, some advantages in using variable-length step sizes to draw lines.[7] Specifically, the *double- and triple-step line-drawing algorithm* sets either two or three pixels per iteration with the same amount of logic and code complexity as the double-step line-drawing algorithm. In this article, we generalize our findings for the line-drawing algorithm to develop a *double- and triple-step interpolation algorithm* that has similar advantages over the double-step interpolation algorithm.

Double-step algorithm

Before discussing these algorithms in greater detail, we first introduce additional notation used throughout the article. Let $a = \dot{x}_0, \dot{x}_1, \ldots, \dot{x}_n = b$ be the $n + 1$ interpolated values obtained by rounding the numbers from Equation 1 on the interval $[a, b]$. Then $\dot{x}_i$, $0 \le i \le n$, is defined by the following mathematical expression:

0272-17-16/$4.00 © 1994 IEEE

$$\dot{x}_i = \lfloor a + ik + 0.5 \rfloor = \lfloor x_i + 0.5 \rfloor \quad (2)$$

In addition, let

$$X_i = x_{2i} = a + 2ik = a + i(2k) \text{ and} \quad (3)$$

$$\dot{X}_i = \lfloor X_i + 0.5 \rfloor \quad (4)$$

where $i = 0, 1, \ldots, \lfloor n/2 \rfloor$.

In the course of developing the double-step algorithm, Rokne and Rao[4] proved that

$$C \le \Delta_i \dot{X} \le C + 1 \quad (5)$$

$$C' \le \Delta_i \dot{x} \le C' + 1 \quad (6)$$

where $\Delta_i \dot{x} = \dot{x}_i - \dot{x}_{i-1}$, $\Delta_i \dot{X} = \dot{X}_i - \dot{X}_{i-1}$, $C' = \lfloor k \rfloor$, and $C = \lfloor 2k \rfloor$.

By setting $l_i = X_i - \dot{X}_{i-1}$, they note that $\Delta_i \dot{X} = C$ if $l_i < C + 0.5$, and $\Delta_i \dot{X} = C + 1$ if $l_i \ge C + 0.5$. They therefore determine the value of $\Delta_i \dot{X}$ by checking the sign of $l_i - (C + 0.5)$. Since $n > 0$, it follows that $D_i = 2n(l_i - (C + 0.5))$ retains the sign of $l_i - (C + 0.5)$, giving

$$\dot{X}_i = \begin{cases} \dot{X}_{i-1} + C & \text{if } D_i < 0 \\ \dot{X}_{i-1} + C + 1 & \text{if } D_i \ge 0 \end{cases} \quad (7)$$

Of course, if $\Delta_i \dot{X}$ is an even number, say $2m$, then $\dot{x}_{2i-1}$ can be readily determined because $\Delta_{2i-1} \dot{x} = m$. However, when $\Delta_i \dot{X}$ is odd, the following comparison determines the value of $\dot{x}_{2i-1}$:

$$\dot{x}_{2i-1} = \begin{cases} \dot{X}_{i-1} + C' & \text{if } D_i < 2(b-a) - 2n(C - C') \\ \dot{X}_{i-1} + C' + 1 & \text{if } D_i \ge 2(b-a) - 2n(C - C') \end{cases} \quad (8)$$

By subtracting D_i from D_{i+1}, Bao and Rokne[6] prove that the value of the discriminator for the next iteration is

$$D_{i+1} = \begin{cases} D_i + 4(b-a) - 2nC & \text{if } D_i < 0 \\ D_i + 4(b-a) - 2n(C+1) & \text{if } D_i \ge 0 \end{cases} \quad (9)$$

As Equations 7, 8, and 9 show, the double-step algorithm performs many multiplications that are powers of two. These multiplications can be eliminated by performing shifts instead.

B5 must also compute an interpolated point and update the discriminator. Therefore, the comparison in Equation 8 is the only additional work required by the double-step interpolation algorithm. Thus, it uses roughly the same amount of logic as B5 each iteration, while looping half as many times.

Double- and triple-step algorithm

The double- and triple-step line-drawing algorithm can set a third pixel in some of the loop iterations.[7] The algorithm therefore sets three pixels whenever possible and two pixels in the remaining iterations. The double- and triple-step interpolation algorithm takes a similar approach but generalizes the line-drawing findings, which assume that the slope of the line being drawn is less than or equal to one (that is, $0 \le b - a \le n$). This assumption does not necessarily hold for linear interpolation. However, the following theorems show that we can still determine additional points during some iterations of the double-step interpolation algorithm.

Lemma 1. The values of the interpolated points are subject to the following restriction: $C \le \Delta_i \dot{x} + \Delta_{i+1} \dot{x} \le C + 1$.

Proof. The proof is similar to that for Equation 9 in Rokne and Rao,[4] given as Equation 5 here. For brevity, we omit the proof.

Theorem 1. If $C = 2C'$ and $\Delta_i \dot{x} = C' + 1$, then $\Delta_{i+1} \dot{x} = C'$.

Proof. By Equation 6, $\Delta_{i+1} \dot{x}$ must equal either C' or $C' + 1$. Suppose $\Delta_{i+1} \dot{x} = C' + 1$. Then $\Delta_i \dot{x} + \Delta_{i+1} \dot{x} = 2C' + 2 = C + 2$, which contradicts Lemma 1. Therefore, the theorem must hold.

Theorem 2. If $C = 2C' + 1$ and $\Delta_i \dot{x} = C'$, then $\Delta_{i+1} \dot{x} = C' + 1$.

Proof. By Equation 6, $\Delta_{i+1} \dot{x}$ must equal either C' or $C' + 1$. Suppose $\Delta_{i+1} \dot{x} = C'$. Then $\Delta_i \dot{x} + \Delta_{i+1} \dot{x} = 2C' = C - 1$, which contradicts Lemma 1. Therefore, the theorem must hold.

Naturally, when an additional point is interpolated, the discriminator must be adjusted accordingly. We can use arguments similar to those for the line-drawing algorithm[7] to show how the discriminator is updated. That is to say, we can redefine the discriminator in similar terms but have it concern steps of size one instead of size two. We do this by letting $l_i'' = x_i - \dot{x}_{i-1}$. Therefore, $k - 0.5 \le l_i'' < k + 0.5$. Since $C' = \lfloor k \rfloor$, l_i'' is restricted to the following range of values:

$$C' - 0.5 \le l_i'' < C' + 1 + 0.5 \quad (10)$$

Hence, if $l_i'' < C' + 0.5$, then $\Delta_i \dot{x} = C'$. Otherwise, $\Delta_i \dot{x} = C' + 1$. It follows that the discriminator for steps of size one, defined as $D_i'' = 2n(l_i'' - (C' + 0.5))$, retains the sign of $l_i'' - (C' + 0.5)$ and that

$$\begin{aligned} D_i'' &= 2n(l_i'' - (C' + 0.5)) \\ &= 2n(x_i - \dot{x}_{i-1} - (C' + 0.5)) \\ &= 2n(a + [i(b-a)/n] - \dot{x}_{i-1} - (C' + 0.5)) \\ &= 2na + 2i(b-a) - 2n\dot{x}_{i-1} - n(2C' + 1) \end{aligned} \quad (11)$$

Subtracting D_i'' from D_{i+1}'' yields

$$\begin{aligned} D_{i+1}'' - D_i'' &= 2(b-a) - 2n(\dot{x}_i - \dot{x}_{i-1}) \\ &= 2(b-a) - 2n\Delta_i \dot{x} \end{aligned} \quad (12)$$

Therefore, for steps of size one, the values of the interpolated points are

$$\dot{x}_i = \begin{cases} \dot{x}_{i-1} + C' & \text{if } D_i'' < 0 \\ \dot{x}_{i-1} + C' + 1 & \text{if } D_i'' \ge 0 \end{cases} \quad (13)$$

and the values of the discriminator are

$$D''_{i+1} = \begin{cases} D''_i + 2(b-a) - 2nC' & \text{if } D''_i < 0 \\ D''_i + 2(b-a) - 2n(C'+1) & \text{if } D''_i \geq 0 \end{cases} \quad \textbf{(14)}$$

Thus, the discriminator is adjusted as shown in Equation 14 when an additional point is interpolated.

Next, we must determine how the algorithm can terminate and still interpolate the correct number of points. Again, we take an approach similar to that for the line-drawing algorithm.[7] Specifically, the loop is exited once $\dot{x}_{n-1}$ is determined. After the algorithm exits the loop, it checks to determine whether the last point, $\dot{x}_n$, has been found.

By using the following theorems and examining all possible cases, we can prove that the algorithm interpolates the correct number of points. For instance, suppose C is even and the last point interpolated in the previous iteration was $\dot{x}_{n-2}$. For the algorithm to interpolate an "extra" point, $\Delta_n\dot{x}$ must equal $C'+1$ (by Theorem 1), which is impossible (by Theorem 3 below). A similar argument holds when C is odd. The proofs for the remaining cases are obvious and are omitted for brevity, although there are some exceptions when $2k$ is integral, which we discuss below.

Theorem 3. If $C = 2C'$, then $\Delta_n\dot{x} = C'$.

Proof. It follows from the symmetric nature of lines and the reduction of the line-drawing problem to linear interpolation that $\Delta_1\dot{x} = \Delta_n\dot{x}$. (The one exception is when $2((b-a) \text{ modulo } n) = n$, that is, $x_1 = q + 0.5$ where q is any integer. This exception cannot occur when $C = 2C'$.) Since the value of $\Delta_1\dot{x} = C'$, the value of $\Delta_n\dot{x}$ must also equal C'.

Theorem 4. If $C = 2C' + 1$ and $2((b-a) \text{ modulo } n) \neq n$, then $\Delta_n\dot{x} = C' + 1$.

Proof. The proof is similar to that for Theorem 3. For brevity, we omit it here.

Implementation

Figure 1 on the next page presents a detailed algorithm that interpolates points whether or not $(b-a) < n$. Note that the double- and triple-step interpolation algorithm has as many cases as the double- and triple-step line-drawing algorithm even though the interpolation algorithm does not require $(b-a)$ to be less than or equal to n; further, the interpolation algorithm must be reversible. The code assumes that the div operator divides two integers, rounding towards zero. In addition, the mod operator returns the remainder that occurs when one number is divided by another. Due to this rounding, the values of C and C' are decremented when $a > b$ (see Figure 1).

Unless some additional checks are made, the algorithm will calculate values of C and/or C' incorrectly when $2k$ is a negative integer. Even though these values could be corrected, we leave them unchanged because, as the following argument shows, the algorithm will still output the correct interpolated values. (As in other discussions, the argument uses some observations from Rokne and Rao.[4])

We begin by noting that when C is incorrectly calculated, either k is an integer (that is, $2k$ is negative and even), or k is not an integer (that is, $2k$ is negative and odd). For both cases,

$$\begin{aligned} D_1 &= 4(b-a) - n(2C+1) \\ &= 4(b-a) - n[2(2k-1)+1] \\ &= 4(b-a) - [4n(b-a)/n] + 2n - n \\ &= n \end{aligned}$$

Therefore, when k is a negative integer, the fourth "if" clause of the detailed algorithm in Figure 1 is satisfied every iteration because

$$\begin{aligned} D_{i+1} &= D_i + 4(b-a) - 2n(C+1) \\ &= D_i + 4(b-a) - 2n[(2k-1)+1] \\ &= D_i + 4(b-a) - [4n(b-a)/n] + 2n - 2n \\ &= D_i + 0 \end{aligned}$$

and the desired results are produced even though $C \neq \lfloor 2k \rfloor$. When k is not an integer,

$$\begin{aligned} V &= 2(b-a) - 2n(C-C') \\ &= 2(b-a) - 2nC' \\ &= 2(b-a) - 2n\lfloor k \rfloor \\ &= 2(b-a) - 2n[((b-a) - (n/2))/n] \\ &= n \end{aligned}$$

where V is the value in Equation 8 to which the discriminator is compared to determine the middle point. Therefore, the algorithm will again work correctly because the second "if" clause of the detailed algorithm is satisfied every iteration.

Another exception occurs when $2k$ is an odd integer and $a < b$, since an extra point can be output. However, the problem is easily corrected by decrementing the value of C (the proof of its correctness follows from the preceding discussion). Hence, the algorithm works correctly when $2k$ is a negative integer or when $2k$ is a positive odd integer, even though $C \neq \lfloor 2k \rfloor$.

Efficiency

We now compare the performance of the double- and triple-step interpolation algorithm with the double-step and the B5 algorithms. (We make no comparison with the A3 algorithm because it can lose accuracy and its speed depends on whether the processor has a barrel shifter.)

To ensure that exactly n points are interpolated when $4(b-a) \leq n$, the algorithm must keep a count of the number of points produced. The following comparisons therefore use a version of the double-step interpolation algorithm slightly modified to this purpose. (Implementing this count has the added benefit of reducing code complexity.) Furthermore, we consider only values of a, b, and n such that $0 \leq (b-a) \leq n/2$. Analysis of the algorithms for the remaining sets of numbers involve similar arguments.

Figure 1. The double- and triple-step incremental linear interpolation algorithm.

```
procedure INTERPOLATION(a, b, n: integer);
int dx, dx2, n2, C, C', C1', D, V;
int incr1, incr2, incr3, i, x, endpt;

begin
  dx = b - a;
  dx2 = 2*dx;
  n2 = 2*n;
  if (dx < 0) then
    begin
      C1' = dx div n;
      C' = C1' - 1;
      C = (dx2 div n) - 1;
    end
  else
    begin
      C' = dx div n;
      C1' = C' + 1;
      if (2*(dx mod n) = n) then C = (dx2 div n)- 1
      else C = dx2 div n;
    end;
  D = 2*dx2 - n2*C - n;
  V = dx2 - n2*C + n2*C';
  incr1 = 2*dx2 - n2*C;
  incr2 = incr1 - n2;
  x = a;
  i = 0;
  endpt = n - 1;
  output(a);
  {Case 1: C is even}
  if (C is even) then
    begin
      incr3 = incr2 + dx2 - n2*C';
      while (i < endpt) do
        {if clause 1: Δ_i ẋ + Δ_{i+1} ẋ = C}
        if (D < 0) then
          begin
            x = x + C';
            output(x);
            x = x + C';
          output(x);
          D = D + incr1;
          i = i + 2;
          end
        {if clause 2: Δ_i ẋ + Δ_{i+1} ẋ = C + 1 and
        Δ_i ẋ = C' + 1}
        else if (D ≥ V) then
          begin
            x = x + C1';
            output(x);
            x = x + C';
            output(x);
            D = D + incr2;
            i = i + 2;
          end
        {if clause 3: Δ_i ẋ + Δ_{i+1} ẋ = C + 1 and Δ_i ẋ =
C'}
        else
          begin
            x = x + C';
            output(x);
            x = x + C1';
            output(x);
            x = x + C';
            output(x);
            D = D + incr3;
            i = i + 3;
          end;
      end

    {Case 2: C is odd}
    else
      begin
        incr3 = incr1 + dx2 - n2*C' - n2;
        while (i < endpt) do
          {if clause 4: Δ_i ẋ + Δ_{i+1} ẋ = C + 1}
          if (D ≥ 0) then
            begin
              x = x + C1';
              output(x);
              x = x + C1';
              output(x);
              D = D + incr2;
              i = i + 2;
            end
          {if clause 5: Δ_i ẋ + Δ_{i+1} ẋ = C and Δ_i ẋ = C'}
          else if (D < V) then
            begin
              x = x + C';
              output(x);
              x = x + C1';
              output(x);
              D = D + incr1;
              i = i + 2;
            end
          {if clause 6: Δ_i ẋ + Δ_{i+1} ẋ = C and Δ_i ẋ = C' + 1}
          else
            begin
              x = x + C1';
              output(x);
              x = x + C';
              output(x);
              x = x + C1';
              output(x);
              D = D + incr3;
              i = i + 3;
            end;
      end
    if (i < n) then output(b);
  end;
```

For our algorithm, $n/2$ iterations occur in the worst case, such as when $(b - a) = 0$. The best-case performance occurs when $3(b - a) = n$. Under this condition, our algorithm interpolates three points each iteration. Thus, it performs one-third fewer iterations than the double-step algorithm, which always determines two points each iteration.

For an average-case analysis, we first note that the relative speed of the double- and triple-step algorithm depends on the value of $(b - a)$. In other words, when $(b - a) = 0$, the algorithm takes two steps every iteration. When $(b - a) = 1$, there can be at most one iteration where three steps are taken, and so on. Assuming that the number of steps of size three is $(b - a)/2$ on av-

Table 1. The number of tests and additions made by the B5, double-step (DS), and the double- and triple-step (DTS) interpolation algorithms for the average case.

	B5	DS	DTS
Tests	$32n/16$	$20n/16$	$18n/16$
Additions	$48n/16$	$32n/16$	$30n/16$

erage and that the average value of $(b - a)$ is $n/4$, the average number of iterations having steps of size three equals $n/8$, and the average number of iterations is $7n/16$. Since the double-step algorithm always iterates $n/2$ times, our interpolation algorithm reduces the number of iterations by 12.5 percent on average. As stated earlier, the B5 algorithm requires n iterations because it always takes one step.

Now that we have determined the number of iterations performed by each algorithm, we can find the amount of work done by the algorithms for the average case: Each iteration of the B5 algorithm performs two tests (one loop control test and one test on the discriminator) and three additions (one addition to calculate the point, one to update the discriminator, and one to increment the count of the points produced). Therefore, B5 requires $2n$ tests and $3n$ additions on average.

On the other hand, the double-step algorithm performs the same two tests as B5 each iteration plus $(b - a)$ additional tests (which equals $n/4$ on average) since the test in Equation 8 is sometimes required. The double-step algorithm also requres four additions (two additions to calculate the two points, one to update the discriminator, and one to increment the count). Therefore, $2(n/2) + n/4 = 5n/4$ tests and $4(n/2)$ additions are performed on average.

For the double- and triple-step algorithm, the work needed in each iteration is similar to that for the double-step algorithm. Thus, it requires $2(7n/16) + n/4 = 18n/16$ tests and $n + 2(7n/16) = 30n/16$ additions (n additions for determining the points plus the additions for updating the discriminator and incrementing the count).

Table 1 summarizes the efficiency comparisons. Our analysis confirms the empirical comparisons made in Rokne and Rao,[4] which state that the ratio of the number of tests made by the double-step and the B5 algorithms equals 0.63, and the ratio of the number of additions equals 0.75 when the additions to update the count are ignored. Although the multiple-step algorithms perform substantially less work than B5 for larger values of n, B5 has less overhead; so it may be desirable to test the step count and use B5 for small n.

Concluding remarks

Our method of linear interpolation generalizes the findings of a variable-step line-drawing algorithm. The resulting interpolation algorithm has as many loops as the line-drawing algorithm, but fewer restrictions on its input variables. Furthermore, its benefits over the fixed-step interpolation algorithms are similar to those of the variable-step line-drawing algorithm. That is, the double- and triple-step interpolation algorithm can reduce the number of iterations of the double-step interpolation algorithm while keeping the code complexity, initialization costs, and worst-case performance the same. The improvement in speed over the single-step B5 algorithm is even greater. ❑

Acknowledgments

This research is supported in part by the Office of Naval Research grant N00014-91-J-1306.

References

1. D. Field, "Incremental Linear Interpolation," *ACM Trans. Graphics*, Vol. 4, No. 1, Jan. 1985, pp. 1-11.
2. T.R.H. Sizer, *The Digital Differential Analyser*, Chapman and Hall, London, 1968.
3. J.E. Bresenham, "Incremental Line Compaction," *Computer J.*, Vol. 25, No. 1, Feb. 1982, pp. 116-120.
4. J. Rokne and Y. Rao, "Double-Step Incremental Linear Interpolation," *ACM Trans. Graphics*, Vol. 11, No. 2, Apr. 1992, pp. 183-192.
5. X. Wu and J. Rokne, "Double-Step Incremental Generation of Lines and Circles," *Computer Vision, Graphics, and Image Processing*, Vol. 37, No. 3, Mar. 1987, pp. 331-344.
6. P. Bao and J. Rokne, "Quadruple-Step Line Generation," *Computers & Graphics*, Vol. 13, No. 4, 1989, pp. 461-469.
7. P. Graham and S.S. Iyengar, "Double- and Triple-Step Incremental Generation of Lines," *Proc. 1993 ACM Computer Science Conf.*, ACM Press, New York, 1993.

Phil Graham is currently employed by Boss Film Studios as a computer-generated imagery programmer, working on special effects and image processing in the entertainment business. His research interests are in the design and analysis of graphic algorithms, particularly fractals, rasterization algorithms, and rendering techniques. Graham received his BS and PhD in computer science from Louisiana State University in Baton Rouge.

S. Sitharama Iyengar is chair of the Computer Science Department and professor of computer science at Louisiana State University. His research interests are in high-performance algorithms and data structures. Iyengar received his BS in engineering from Bangalore University, India, his MS from the Indian Institute of Science, and his PhD from Mississippi State University. He is an ACM national lecturer, a series editor for *Neuro Computing of Complex Systems*, and area editor for the *Journal of Computer Science and Information*.