Fast Image Transforms Using Diophantine Methods
Sharat Chandran, Member, IEEE, Ananth K. Potty, and Milind Sohoni

Abstract—Many image transformations in computer vision and graphics involve a pipeline when an initial integer image is processed with floating point computations for purposes of symbolic information. Traditionally, in the interests of time, the floating point computation is approximated by integer computations where the integerization process requires a guess of an integer. Examples of this phenomenon include the discretization interval of $\rho$ and $\theta$ in the accumulator array in classical Hough transform, and in geometric manipulation of images (e.g., rotation, where a new grid is overlaid on the image).

The result of incorrect discretization is a poor quality visual image, or worse, hampers measurements of critical parameters such as density or length in high fidelity machine vision. Correction techniques include, at best, anti-aliasing methods, or more commonly, a "kludge" to cleanup. In this paper, we present a method that uses the theory of basis reduction in Diophantine approximations; the method outperforms prior integer based computation without sacrificing accuracy (subject to machine epsilon).

Index Terms—Basis reduction, floating point arithmetic, integer computations, lattice, simultaneous.

I. INTRODUCTION

A large number of problems in computer vision involve floating point computations in image manipulation. Typically, the input image is a quantized version of the original analog signal and, therefore, stored as a byte image. A transformation is applied to process the image; this process may result in intermediate floating point numbers. However, the final image displayed or analyzed is similar to the input image—stored as a byte image. As an example, consider the following problem which we use as a motivation.

A. An Example

A three-dimensional (3-D) image is available to us as an integer matrix $[N \cdots N][-N \cdots N][-N \cdots N]$. The location $image[\text{1}[\text{1}[\text{1}]]$ stores, for example, the gray value at the point $i\vec{x} + j\vec{y} + k\vec{z}$, where $\vec{x}$, $\vec{y}$, $\vec{z}$ are orthogonal unit vectors in 3-D space parallel to the $x$, $y$, and $z$ Cartesian axes respectively.

Now consider a rotation (characterized by the matrix $M_{rot}$) such that the orthonormal basis $[\vec{x}, \vec{y}, \vec{z}]$ goes to the new basis $[\vec{x}^\prime, \vec{y}^\prime, \vec{z}^\prime]$. The new basis, in terms of the old basis is given as

$$ [\vec{x}^\prime \ \vec{y}^\prime \ \vec{z}^\prime] = M_{rot} [\vec{x} \ \vec{y} \ \vec{z}] .$$

If, as assumed here, $\vec{x} = [1, 0, 0]^T$, then $\vec{x}^\prime$ is obtained from the first column of the rotation matrix $M_{rot}$.

The formulation in this paper is somewhat general. It may be applied when we perform animation, and we have repeated use of the rotation matrix in which case the right hand side represents an arbitrary vector instead of one parallel to the Cartesian axes. The formulation also does not use the orthogonality of the rotation matrix.

We would now like to assign gray values to the new rotated image, i.e., the integer matrix $\text{rotated_image}[-N \cdots N][-N \cdots N][-N \cdots N]$ (notice that the output is also of the integer data-type, and we ignore, for the moment, that the size of the rotated image might be different from the given image). Fig. 1 shows the basis vectors when the space is rotated by an angle $\kappa$ about the $Z$ axis.

To rotate the image, we have to rotate every point $(i, j, k)$. $i$, $j$, $k = -N, \ldots, N$. This entails $O(N^2)$ floating point matrix multiplications as seen below which is linear in the input size.

B. An Intuitive Method

A simple method to rotate the image is to obtain the point $v[i \cdots 3]$ (in the given image) associated with position $(i, j, k)$ in the rotated image as follows (for each grid position in the new image, we look for the correct position in the old image to obtain the gray value)

$$ v = [i \ j \ k] \begin{bmatrix} x^T \\ y^T \\ z^T \end{bmatrix} .$$

Fig. 1. Rotation of image space about the $Z$ axis. The vector $\vec{x}^\prime$ is a unit vector along the direction shown as XX. The gray value at the point $P^\prime$ is the same as the gray value at $P$. 
Here, we obtain a $1 \times 3$ matrix in the matrix product of a $1 \times 3$ matrix with a $3 \times 3$ matrix.

Once we have the vector $v$, we obtain the integer vector $\text{trunc}_v[1\ldots3]$ by truncating the values of the floating point vector $v$. Thus the final code may look like Algorithm 1 which costs $9N^2$ floating point multiplications.

Algorithm 1 Pseudo Code for the Intuitive Method. trunc() is the Normal Truncation Operation.

```
1: Procedure rotate ($xx, yy, zz$: array $[1\ldots3]$ of integer)
2:   for $i := -N$ to $N$
3:       for $j := -N$ to $N$
4:         for $k := -N$ to $N$
8:           $\text{trunc}_v[1] := \text{trunc}(v[1])$;
10:          $\text{trunc}_v[3] := \text{trunc}(v[3])$;
11:         rotated_image[$i$][$j$][$k$] :=
13:     end for
14: end for
15: end for
16: end Procedure
```

C. Drawbacks

Define transform_index to be the coordinates of the point $P$ with respect to the coordinates $(i, j, k)$ in the new (rotated) basis, i.e., $\text{trunc}_v$ at $(i, j, k)$. (This quantity need not be explicitly computed, and may be expensive in terms of memory if explicitly stored.)

1) For any integer tuple $(i, j, k)$, the vector transform_index[$i$][$j$][$k$] is again an integer vector. However the computations are performed using floating point arithmetic. Floating point computations are expensive as compared to integer computations, and hence computing all the values of transform_index[$i$][$j$][$k$] would consume a fair amount of time.

2) A more subtle drawback is that a lot of the computations in Algorithm 1 are repetitive and therefore redundant—as we shall see, a new value for a particular tuple can be obtained from earlier computed values.

II. RESULTS ACHIEVED AND EARLIER WORK

In this document we present a method by which the $O(N^3)$ floating point matrix multiplications of this problem can be replaced by $O(N^3)$ integer matrix additions. There are, therefore, twin rewards. Floating point calculations are replaced by integer ones, and multiplications are replaced by additions. We also guarantee that the resulting values are correct; no loss of accuracy results in our process in going from one integer domain to another integer domain.

More generally, we observe that the rotated image is a digitized image and hence the result of the floating point computations has to be truncated before the gray values are assigned to the output image. This motivates us to explore the possibility of obtaining the final result through largely integer computations in a variety of situations when the net effect of the processing pipeline is an integer.

Examples of such situations, in which our methods are useful, include the elimination of floating point computations in generic image manipulations (that is, other than pure rotations), in image compression in the JPEG computation of the Discrete Cosine Transform (see, for example, [7]), in the classic computation of optical flow [6], in edge relaxation as in [8], in the discretization of the accumulator array in classical Hough Transform [1], and, indeed, in a variety of situations.

There have been several methods for achieving fast rotations, indeed, most of this knowledge is in textbooks and programming folklore. A common technique is to rewrite the rotation matrix to use shearing transforms [9]. While this approach is fast, it does not guarantee accuracy, as it involves floating point computations. An alternative is to use the FFT approach (for example, [2]) especially if the input is given in the Fourier domain; however, now the floating point errors become worse due to the process of performing computations in the frequency domain, and then returning back to the time domain.

Conceptually, the Cordic algorithm [5], [10] resembles our approach because the attempt is made to perform exact computation, as well as avoid floating point arithmetic. This may be necessary when a hardware multiplier is unavailable (e.g., in a microcontroller) or when we want to save the gates required to implement one (e.g., in an FPGA). Cordic achieves precision in theory only after an unbounded number of iterations. In contrast, in theory our algorithm is provably convergent in a small polynomial in the number of bits to represent the biggest number as input (for example, the dimensions of an input array); in practice, the algorithm converges in about 4 or 5 steps. Cordic is particularly tailored to rotations and relies on specific trigonometric identities. Our work makes no assumption on the floating point numbers with which the transformed index is obtained; in this situation, Cordic cannot be applied.

Methods to achieve exact results include the techniques of interval arithmetic (where the emphasis is to go beyond the machine epsilon), and rational arithmetic. However, all of these methods sacrifice speed for accuracy. Our method is as accurate as the intuitive method mentioned above, but is also fast.

Subsequent layout of this paper is as follows. In the next section, we provide other attempts to give integer solutions, and the pitfalls therein. In Section IV, we present our proposal, and finally the last section gives representative results.

III. INTEGER SOLUTIONS

A representative scheme for the problem results in a solution that performs only $O(N)$ floating point computations, and $O(N^3)$ integer computations. Unfortunately, this widespread method results in incorrect values.
A. Representative Solution

In essence, the method involves precomputing the vectors \( D_{xx}, D_{yy}, D_{zz} \); array \([-N \cdots N]\) of array \([1 \cdots 3]\) of integer; by the code fragment given in Algorithm 2.

**Algorithm 2**

**Pseudo Code for the Integer Solution That Uses \( O(N) \) Floating Point Computations.**

1: for loopVar := -N to N do
2: \( p := D \times \text{loopVar}; \)
3: \( D_{xx}[\text{loopVar}] := \text{trunc}(p \times xx[1]); \)
4: \( D_{xx}[\text{loopVar}] := \text{trunc}(p \times xx[2]); \)
5: \( D_{xx}[\text{loopVar}] := \text{trunc}(p \times xx[3]); \)
6: \( D_{yy}[\text{loopVar}] := \text{trunc}(p \times yy[1]); \)
7: \( D_{yy}[\text{loopVar}] := \text{trunc}(p \times yy[2]); \)
8: \( D_{yy}[\text{loopVar}] := \text{trunc}(p \times yy[3]); \)
9: \( D_{zz}[\text{loopVar}] := \text{trunc}(p \times zz[1]); \)
10: \( D_{zz}[\text{loopVar}] := \text{trunc}(p \times zz[2]); \)
11: \( D_{zz}[\text{loopVar}] := \text{trunc}(p \times zz[3]); \)
12: end for

Here, \( D \) is a suitably chosen integer (say 16). Thus, in effect, \( D_{xx}, D_{yy}, D_{zz} \) are approximations of the integral multiples of the floating point vectors \( xx, yy \) and \( zz \) as rational numbers, with denominators \( D \).

Computation of \( \text{transform\_index}[i][j][k] \) is done using purely integer calculations as follows:

\[
\text{ix}[1] := (D_{xx}[i][1] + D_{yy}[j][1] + D_{zz}[k][1]) \div D;
\]
\[
\text{ix}[2] := (D_{xx}[i][2] + D_{yy}[j][2] + D_{zz}[k][2]) \div D;
\]
\[
\text{ix}[3] := (D_{xx}[i][3] + D_{yy}[j][3] + D_{zz}[k][3]) \div D
\]
and rotated_image\([i][j][k]\) is computed using the following code fragment:

\[
\text{rotated\_image}[i][j][k] := \text{image}[\text{ix}[1][\text{ix}[2]][\text{ix}[3]]
\]

and is repeated for all \( O(N^3) \) tuples.

The division by \( D \) above is integer division (performed using shifts if \( D \) is a power of 2), and offsets the earlier multiplication by \( D \) in computing \( D_{xx}, D_{yy}, D_{zz} \).

B. Idea Behind the Representative Solution

As shown in Fig. 2, the above method attempts to obtain a fraction \( p/q \) such that for most integers \( i < N \), truncating \( xx \) and \( iy/p \times q \) yield the same integer. \( q = 16 \) in Section III-A and \( p \) changes with the loop variable index.

C. Discussion

Clearly, the vector \( iv \), differs from the ideal \( \text{transform\_index}[i][j][k] \) of Section I-C leading to some amount of distortion in the image (see Fig. 3). This can be easily verified by computing the values for \( xx[1] = 0.846154, yy[1] = 0.714286, zz[1] = 0.600000 \). Table I illustrates some examples (for \( D = 16 \) and \( N = 256 \)). In other words, an incorrect \( p/q \) results due to the compounded errors in the approximations of \( D_{xx}, D_{yy}, D_{zz} \). This may be critical when, for instance, medical images are used in diagnostic applications, or in nano measurement applications, or even in common 3-D rendering of animated scenes.

Table I

**CORRESPONDING IMAGE POSITIONS AS COMPUTED BY REPRESENTATIVE INTEGER SOLUTIONS, AND THE CORRECT METHOD (ALGORITHM 1). A STAR INDICATES A “DON’T CARE” VALUE. IT IS NOT COMPUTED HERE SIMPLY TO ILLUSTRATE THE ERROR IN ONE DIMENSION**

<table>
<thead>
<tr>
<th>(i, j, k)</th>
<th>by Algorithm 2</th>
<th>by Algorithm 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 2, 1)</td>
<td>( \text{ix}[1][\text{ix}[2]] )</td>
<td>( \text{ix}[1][\text{ix}[2]] )</td>
</tr>
<tr>
<td>(1, 2, 3)</td>
<td>( \text{ix}[1][\text{ix}[2]] )</td>
<td>( \text{ix}[1][\text{ix}[2]] )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

There are two arguments to the drawbacks mentioned:

- The image distortion resulting from the above is offset by the benefits of speedy computation.
- As \( D \) increases, the error in the values is seen only for large values of \( i, j, \) and \( k \). If \( D = 100000 \) in the above example, then it is possible to verify that the above method and the real computations cannot differ.

The response to the second argument is that choosing \( D = 100000 \) is valid only for the specific example in Table I. Further, \( D \) should not be extremely large to avoid memory overflows; finally, there is some amount of guesswork involved in determining \( D \) for different sized images.

The response to the first is more subtle. Other than the obvious problem of using incorrect values, there are various algorithms in which “missing” a value is critical. For example, in volumetric visualization, when rays are shot and projected onto a plane, “missing” a voxel can lead to a completely different...
voxel being projected on to the screen. A “red” color in an otherwise blue background can cause considerable visualization discomforts.

In the next section we introduce a method by which the approximation of \( v_i \), (i.e., \( i \nu \) or transform_index[k][j][k] ), is exactly the same as that for the real valued computations shown in Section I-C, and the guesswork involved in determining the right \( D \) is eliminated. The extra penalty to be paid for this is not much in terms of running time, and in fact, paves the way to an even faster algorithm.

IV. OUR PROPOSAL

Our proposed solution to the above problem is based on the applications of the theory of basis reduction in Diophantine Approximation [4]. We define a new problem formally and show later that the solution to this problem results in a method that completely eliminates floating point computations, yet, maintains the desired accuracy.

A. The Problem

The basic problem addressed by the above methods is the following. Suppose we are given numbers \( r_1 \), \( r_2 \) and \( r_3 \) and an integer \( N \). Is it possible to approximate \( r_1 \), \( r_2 \) and \( r_3 \) by fractions \( p_0/q \), \( p_2/q \) \( p_3/q \) with the same denominator \( q \), such that for all \( i, j, k \) in the interval \(-N, \ldots, N\)

\[
\text{trunc}(ir_1 + jr_2 + kr_3) = \text{trunc} \left( i \frac{p_0}{q} + j \frac{p_2}{q} + k \frac{p_3}{q} \right). \tag{1}
\]

We term this as the problem of Simultaneous Diophantine Approximation (SDA). There is an efficient procedure which on input \( r_1, \ r_2, \ r_3, \) and \( N \) will output the rational approximation with the above required property (see Theorem 2)—the time taken by the algorithm is proportional to the number of bits needed to express \( N \). Section IV-B shows why the result is interesting to us. But first, we need the following result.

Theorem 1: There exists an algorithm that, given numbers \( \alpha_1, \ldots, \alpha_n \) and \( 0 \leq \epsilon < 1 \), computes SDA approximation integers \( p_1, \ p_2, \ldots, p_n \), and an integer \( q \) such that

\[
1 \leq q \leq 2^n(n+1)/4\epsilon^{-n}
\]

and

\[
|\alpha_i q - p_i| < \epsilon \quad (i = 1, \ldots, n).
\]

Proof: See [4].

To show that the approximations found using Theorem 1 are indeed the approximations we are looking for, we present the following theorem.

Theorem 2: Let \( r_1, r_2 \) and \( r_3 \) be any given numbers, \( N \in \mathbb{Z} \), \( 0 \leq \epsilon < 1/3N \) and \( p_0/q, \ p_2/q \) and \( p_3/q \) be their SDA approximations, then \( \exists z \in \mathbb{Z} \) such that

\[
ir_1 + jr_2 + kr_3 \leq i \frac{p_0}{q} + j \frac{p_2}{q} + k \frac{p_3}{q}, \quad i, j, k \leq N \tag{2}
\]

or

\[
ir_1 + jr_2 + kr_3 > i \frac{p_0}{q} + j \frac{p_2}{q} + k \frac{p_3}{q}, \quad i, j, k \leq N. \tag{3}
\]

Proof: Since \( p_1/q, \ p_2/q \) and \( p_3/q \) are the Simultaneous Diophantine Approximations of \( r_1, \ r_2 \) and \( r_3 \) we have, from Theorem 1

\[
|qr_i - p_i| < \epsilon, \quad i = 1, 2, 3, \tag{4}
\]

Assume, if possible, \( \exists z \in \mathbb{Z} \) such that for some \( i, j, k \leq N \)

\[
ir_1 + jr_2 + kr_3 < i \epsilon + j \epsilon + k \epsilon
\]

or

\[
ir_1 + jr_2 + kr_3 > i \epsilon + j \epsilon + k \epsilon.
\]

Subtracting, we get

\[
0 < zq - p_0 - p_2 - p_3 \leq i(qr_1 - p_1) + j(qr_2 - p_2) + k(qr_3 - p_3)
\]

and

\[
0 < zq - p_0 - p_2 - p_3 < 3N \epsilon
\]

resulting in a contradiction that there exists an integer between 0 and 1. Hence \( \exists z \in \mathbb{Z} \) satisfying (3). A similar argument implies that \( \exists z \in \mathbb{Z} \) satisfying (2). Q.E.D.

B. Applying the Approximations

We solve three SDA problems for the tuples \( (xx[1], \ yy[1], \ zz[1]), \ (xx[2], \ yy[2], \ zz[2]), \) and \( (xx[3], \ yy[3], \ zz[3]) \). Each SDA problem involves about 4 or 5 steps working on a matrix of size \( 4 \times 4 \) of floating point numbers. Having obtained the rational simultaneous approximations \( p_{xx[a]/q[a]},\ p_{yy[a]/q[a]}, \) and \( p_{zz[a]/q[a]} \), \( a = 1, 2, 3 \), the computation of \( xx[a] \) is accomplished completely in integer arithmetic as follows.

We may optionally precompute and store \( D_{xx}, \ D_{yy}, \) and \( D_{zz} \) as described in Algorithm 3.

Algorithm 3 Proposed Pseudo Code

1: SDApproximate the triples \( (xx[1], \ yy[1], \ zz[1]), \ (xx[2], \ yy[2], \ zz[2]), \) \( (xx[3], \ yy[3], \ zz[3]) \) to obtain \( (p_{xx[1]}, \ p_{yy[1]}, \ p_{zz[1]}), \ (q[1]), \) \( (p_{xx[2]}, \ p_{yy[2]}, \ p_{zz[2]}), \ (q[2]), \) \( (p_{xx[3]}, \ p_{yy[3]}, \ p_{zz[3]}), \) and \( q[3] \). (see Algorithm 5)

2: for loopVar := -N to N do

3: \( D_{xx}[loopVar[1]] := \text{loopVar} \cdot p_{xx[1]}; \)

4: \( D_{xx}[loopVar[2]] := \text{loopVar} \cdot p_{xx[2]}; \)

5: \( D_{xx}[loopVar[3]] := \text{loopVar} \cdot p_{xx[3]}; \)

6: \( D_{yy}[loopVar[1]] := \text{loopVar} \cdot p_{yy[1]}; \)

7: \( D_{yy}[loopVar[2]] := \text{loopVar} \cdot p_{yy[2]}; \)

8: \( D_{yy}[loopVar[3]] := \text{loopVar} \cdot p_{yy[3]}; \)

9: \( D_{zz}[loopVar[1]] := \text{loopVar} \cdot p_{zz[1]}; \)

10: \( D_{zz}[loopVar[2]] := \text{loopVar} \cdot p_{zz[2]}; \)

11: \( D_{zz}[loopVar[3]] := \text{loopVar} \cdot p_{zz[3]}; \)

12: end for


Now by Theorem 2, the vector \( \hat{v} \) obtained by
\[
\hat{v}[1] := (D_{xx}[i][1] + D_{yy}[j][1] + D_{zz}[k][1]) \mod q[1];
\hat{v}[2] := (D_{xx}[i][2] + D_{yy}[j][2] + D_{zz}[k][2]) \mod q[2];
\hat{v}[3] := (D_{xx}[i][3] + D_{yy}[j][3] + D_{zz}[k][3]) \mod q[3];
\]
matches, in the sense of (1) the ideal vector \( \text{transform}_i[j][k] \) (from Section I-C). At this point, if we ignore the time for the SDA approximation, we have an algorithm that uses \( 9N \) integer multiplications in a preprocessing stage, and at run time uses \( 3N^3 \) integer divisions to fill the output array of size \( N^3 \). The preprocessing requires a memory storage of size \( N^9 \).

C. Eliminating Integer Divisions

We have seen in the previous section a technique by which we can reduce floating point multiplication to integer computations. Here we note that the division by an integer still incurs a major cost. In this section we propose a method by which this cost can be eliminated.

We observe that in our original problem, we rotate every point in the raster grid. We exploit the fact that the position in \( \text{transform}_i[j][k] \) is related to the positions in the neighboring cells.

This is seen from the following equations (where \( a \) is instantiated to 1, 2, or 3):
\[
\begin{align*}
\text{transform}_i[j][k][a] &= (D_{xx}[i][a] + D_{yy}[j][a] + D_{zz}[k][a]) \mod q[a]; \\
\text{transform}_i[j][k][a] &= \text{transform}_i[j][k-1][a] + p_{zz}[a] \mod q[a].
\end{align*}
\]

Assuming without loss of generality that \( p_{zz}[a] \) is less than \( q[a] \), i.e., the number \( x \) which was approximated to obtain the fraction \( p_{zz}[a]/q[a] \) is less than 1. Instead of computing the last division explicitly, we keep track of the numerator and denominator using the notion of a residue that can attain only two values (namely, 0 and 1). When the numerator exceeds \( q[a] \), residue attains the value 1, and we reset the counter. The details of this residue arithmetic is shown in Algorithm 4. (A tricky point in this memorization technique is that in three dimensions, we need residue to be two-dimensional in nature.) Algorithm 4 uses about \( 3N^3 \) elementary operations involving comparisons and cached additions.

Algorithm 4 Eliminating Integer Division

```plaintext
1: Procedure mapVoels(\( p_{xx}, p_{yy}, p_{zz}, q \): integer; comp: 1...3)
2: var residue: array[0...N][0...N] of integer;
3: begin
4: transform_index[0][0][0][comp] := 0;
5: residue[0][0] := 0;
6: for \( i := 1 \) to \( N \) do
7: \[ residue[i][0] := residue[i-1][0]+p_{xx}; \]
8: if \( \text{residue[i][0]} < q \) then
9: \[ \text{transform_index}[i][0][0][\text{comp}] := \text{transform_index}[i-1][0][0][\text{comp}]; \]
10: \[ \text{transform_index}[i][0][0][\text{comp}] := \text{transform_index}[i-1][0][0][\text{comp}]+1; \]
11: end if
12: for \( j := 1 \) to \( N \) do
13: if \( \text{residue[i][j]} < q \) then
14: \[ \text{transform_index}[i][j][0][\text{comp}] := \text{transform_index}[i][j-1][0][\text{comp}]; \]
15: \[ \text{transform_index}[i][j][0][\text{comp}] := \text{transform_index}[i][j-1][0][\text{comp}]+1; \]
16: end if
17: for \( k := 1 \) to \( N \) do
18: \[ \text{transform_index}[i][j][k][\text{comp}] := \text{transform_index}[i][j][k-1][\text{comp}]; \]
19: \[ \text{transform_index}[i][j][k][\text{comp}] := \text{transform_index}[i][j][k-1][\text{comp}]+1; \]
20: end if
21: end for
22: end for
23: end procedure
```

V. Results and Concluding Remarks

Many image transformations in computer vision involve a pipeline when an initial integer image is processed with floating point computations for purposes of symbolic information. Traditionally, in the interests of time, the floating point computation is approximated by integer computation where the integerization process requires a guess of a magic integer. This results in poor quality image (which is often “cleaned” using a low pass filter).

In this paper, we have shown a faster (than traditional) method to solve a specific instance of a family of image related computations, namely, that of 3-D image rotation. It is surprising to note that in the process, as we see below, we improve both speed and quality. We document our results for here. Firstly, Figs. 4 and 5 show the results obtained from using Algorithms 2 (a traditional method) and 4 (our proposed method) to rotate the 3-D image of a skull by angles 1.0, 1.0 and 33 degrees about the
Fig. 4. Result obtain using naive integer approximations (Algorithm 2).

Fig. 5. Results obtained using Algorithm 4 is visibly different.

$X$, $Y$, and $Z$ axes respectively. We observe that the image obtained by Algorithm 2 is different from the correct image as also shown numerically as in Table I. This is because of the error in the integerization in the calculation of the position of the point in the rotated space.

To eliminate errors in the discretization process, slow floating point computations may be adopted; our method shows that speed need not be sacrificed for quality. We implemented Algorithms 1, 2, and 4. These algorithms were compared for their performance on various platforms (using the same compiler) and the results are shown (for $N = 256$) in Table II.

The experiments are deliberately stacked against our approach because most of the microprocessors used have hardware floating point processors. The strength of our algorithms is seen that despite this, our proposed method wins in all cases. The win is especially handsome on the Intel 386 processor (which does not have the floating point co-processor).

It has been realized early on that even if sampling theory considerations are employed in the discretization process, image rotation causes aliasing. Anti-aliasing techniques [3] call for the value of an image pixel in the rotated image to be obtained from a number of samples in the original image. Our algorithm, as described in this paper, uses only one sample value in the original image; a better solution would be to use the near neighbors of this sample. This requires a minor modification if we use Algorithm 3, and a less trivial modification to Algorithm 4, since we compute the coordinates of the sample correctly and exactly. Specifically, the equations in Section IV-B will be modified to include the $D$ values of the neighbors.

In analyzing the costs of our algorithm, we note that there is a preprocessing cost of obtaining SDA approximations. The cost of this is about 30 operations on a matrix with 16 floating point numbers. This cost is insignificant when compared to the run time cost of rotating a large volumetric image. Specifically, at run time, if we can afford $9 \times N$ space for storing intermediate values, and $N^2$ space for storing a binary residue matrix, the runtime costs us about $3N^3$ integer operations involving additions and comparisons. This cost (which is linear since we need to read the volumetric array) dominates the preprocessing time for any reasonably sized image. If we cannot afford any extra space (other than the input and output volumetric images), then the runtime cost increases to $3N^3$ integer multiplication operations. The computed answers are exact in any case.

In summary, our method is superior to a floating point based method (employed in the interests of quality), or to a traditional integer-based computations (employed in the interests of time). We have also formally proved that our method is as exact as the floating point based method (subject to machine epsilon in either case).

APPENDIX

SOLVING SIMULTANEOUS DIOPHANTINE EQUATIONS

The algorithm to solve SDA [4] is shown in Algorithm 5; an implementation of this will satisfy (1).

**Algorithm 5** Approximate With Error Less Than Epsilon

1: Procedure SDAApproximate($\alpha[1 \cdots n]$, $\epsilon$: real; $p[1 \cdots n]$, $q$: integer)
2: $B := [\alpha[1 \cdots n] \alpha, 2^{-(n+1)/4}]$
3: $B^* := \text{GramSchmidtOrthogonalize}(B)$
4: loop
5: let $\mu := BB^T \{B^* \text{ is orthogonal}$
6: for $j := 1 \text{ to } n$ do

---

**TABLE II** COMPARISON OF THE PERFORMANCE OF THE ALGORITHMS ON DIFFERENT MACHINES

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time to execute user instructions</th>
<th>% gain of 4 over 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 100</td>
<td>23.34s</td>
<td>174.53%</td>
</tr>
<tr>
<td>Pentium 60</td>
<td>48.71s</td>
<td>112.70%</td>
</tr>
<tr>
<td>Alpha 233</td>
<td>26.33s</td>
<td>34.61%</td>
</tr>
<tr>
<td>SGI R5000</td>
<td>29.13s</td>
<td>194.83%</td>
</tr>
<tr>
<td>Sun 4m</td>
<td>24.79s</td>
<td>64.60%</td>
</tr>
<tr>
<td>Intel 386*</td>
<td>20.99s</td>
<td>358.29%</td>
</tr>
</tbody>
</table>

* $N = 64$
for $i := 1$ to $j - 1$ do
    $B_j := B_{j-\text{round}}(\mu_{j,i})B_i$;
end for

if $\exists j$ such that $\|B_{j+1}^* + \mu_{j+1,j}B_j\| < (3/4)\|B_j\|^2$ then
    swap $B_j$ and $B_{j+1}$;
    adjust $B^*$ accordingly;
else
    break;
end if

end loop

end Procedure

REFERENCES


Sharat Chandran (S’83–M’86) received an undergraduate degree in electrical and electronics engineering from the Indian Institute of Technology, Bombay, and the Ph.D. degree in computer science from the University of Maryland, College Park.

He is currently a Visiting Associate Professor at the University of Maryland and a tenured Associate Professor at the Indian Institute of Technology. He has held appointments in research and engineering at Stanford University, Sタンフォード, CA, Oracle Corporation, NTT (Japan), and Schlumberger. His research interests are in computer graphics and vision, and in parallel computation, fields in which he has made noteworthy contributions in the form of archival journal articles.

Ananth K. Potty received the M.Tech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1998.

He is now the Chief Executive Officer of Retort Soft, India. His current interests are in graphics and vision, networking, and optimizations.

Milind Sohoni received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology (IIT), Bombay, in 1986 and the M.S. degree from the University of Illinois, Urbana-Champaign, in 1988. His doctoral work was from IIT in linear programming.

He is now an Associate Professor at IIT. His current interests are in geometry and CAD/CAM.