Fast image transforms using Diophantine methods

Sharat Chandran
A. K. Potty
M. Sohoni
Department of Computer Science & Engineering
Indian Institute of Technology,
Powai, Mumbai - 400 076. INDIA.
{sharat,apkp,sohoni}@cse.iitb.ac.in

***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 a integer. Examples of this phenomenon include the discretization interval of ρ and θ 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).

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 3-dimensional image is available to us as an integer matrix $image[-N..N][-N..N][-N..N]$. The location $image[i][j][k]$ stores, for example, the grey 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-dimensional 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 $[x, y, z]$ goes to the new basis $[\vec{xx}, \vec{yy}, \vec{zz}]$. The new basis, in terms of the old basis is given as

$$\left[\vec{xx} \quad \vec{yy} \quad \vec{zz}\right] \quad = \quad M_{rot}\left[\vec{x} \quad \vec{y} \quad \vec{z}\right]$$

If, as assumed here, $\vec{x} = [1, 0, 0]^T$, then $\vec{xx}$ 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 grey values to the new rotated image, i.e., the integer matrix rotated_image[-N..N][-N..N][-N..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). Figure 1 shows the basis vectors when the space is rotated by an angle κ about the Z axis.

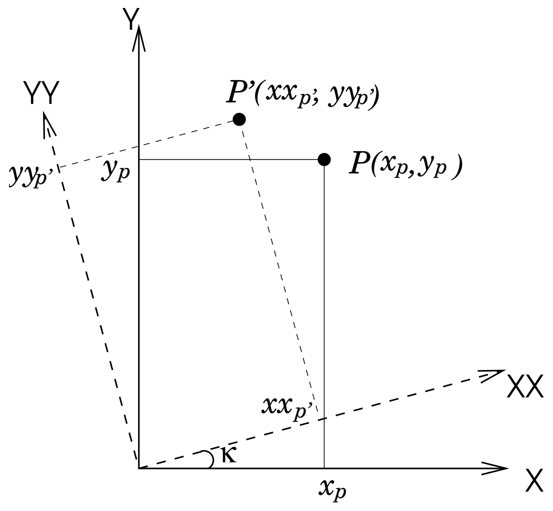

Fig. 1. Rotation of image space about the Z axis. The vector $\vec{xx}$ is a unit vector along the direction shown as XX. The grey value at the point P' is the same as the grey value at P.

To rotate the image, we have to rotate every point $(i, j, k), i, j, k = -N, \ldots, N$. This entails $O(N^3)$ 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 $v[1..3]$ (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),

$$v = \begin{bmatrix} i & j & k \end{bmatrix} \quad \begin{bmatrix} \vec{xx}^T \\ \vec{yy}^T \\ \vec{zz}^T \end{bmatrix}$$

Here we obtain a 1×3 matrix in the matrix product of a 1×3 matrix with a 3×3 matrix.

Once we have the vector v, we obtain the integer vector trunc_v[1..3] by truncating the values of the floating point vector v. Thus the final code may look like Algorithm 1 which costs $9N^3$ floating point multiplications.

Algorithm 1 Pseudo code for the intuitive method. trunc() is the normal truncation operation.

```
Procedure rotate (xx, yy, zz : array [1..3] of integer)
  for i := -N to N do
    for j := -N to N do
      for k := -N to N do
        v[1] := i * xx[1] + j * yy[1] + k * zz[1];
        v[2] := i * xx[2] + j * yy[2] + k * zz[2];
        v[3] := i * xx[3] + j * yy[3] + k * zz[3];
        trunc_v[1] := trunc(v[1]);
        trunc_v[2] := trunc(v[2]);
        trunc_v[3] := trunc(v[3]);
        rotated_image[i][j][k] :=
        image[trunc_v[1]][trunc_v[2]][trunc_v[3]];
      end for
    end for
  end for
end Procedure
```

C. Drawbacks

Define transform_index to be the co-ordinates of the point P with respect to the coordinates (i, j, k) in the new (rotated) basis, i.e., 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 grey 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, [Nel93]), in the classic computation of optical flow [Hor86], in edge relaxation as in [Pra80], in the discretization of the accumulator array in classical Hough Transform [BB82], 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 [TQ97]. 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, [CT99]) 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 [Vol59], [GM95] 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 pre-computing the vectors

D_{xx}, D_{yy}, D_{zz} : array[-N..N] of array[1..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.

```
for loopVar := -N to N do
    p := D $\times$ loopVar;
    $D_{xx}$[loopVar][1] := trunc(p $\times$ xx[1]);
    $D_{xx}$[loopVar][2] := trunc(p $\times$ xx[2]);
    $D_{xx}$[loopVar][3] := trunc(p $\times$ xx[3]);
    $D_{yy}$[loopVar][1] := trunc(p $\times$ yy[1]);
    $D_{yy}$[loopVar][2] := trunc(p $\times$ yy[2]);
    $D_{yy}$[loopVar][3] := trunc(p $\times$ yy[3]);
    $D_{zz}$[loopVar][1] := trunc(p $\times$ zz[1]);
    $D_{zz}$[loopVar][2] := trunc(p $\times$ zz[2]);
    $D_{zz}$[loopVar][3] := trunc(p $\times$ zz[3]);
end for
```

Here D is a suitably chosen integer (say 16). Thus, in effect, D_{xx}, D_{yy} and 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 transform_index[i][j][k] is done using purely integer calculations as follows:

iv[1] := (D_{xx}[i][1] + D_{yy}[j][1] + D_{zz}[k][1]) div D;
iv[2] := (D_{xx}[i][2] + D_{yy}[j][2] + D_{zz}[k][2]) div D;
iv[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:

rotated_image[i][j][k] := image[iv[1]][iv[2]][iv[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} and D_{zz}.

B. Idea behind the representative solution

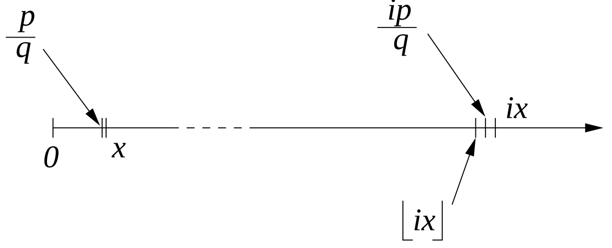

Fig. 2. We want to find $\frac{p}{q}$ such that $\lfloor \frac{ip}{q} \rfloor = \lfloor ix \rfloor$ where x is a number on the real line.

As shown in Figure 2, the above method attempts to obtain a fraction $\frac{p}{q}$ such that for most integers $i < N$, truncating ix and $\frac{ip}{q}$ 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 transform_index[i][j][k] of Section I-C leading to some amount of distortion in the image (see Figure 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 v results due to the compounded errors in the approximations of D_{xx}, D_{yy}, and 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 three-dimensional rendering of animated scenes.

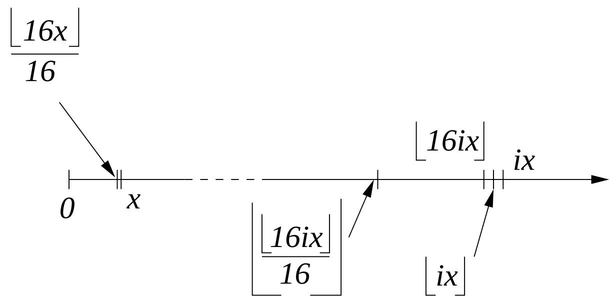

Fig. 3. Error caused due to incorrect choice of D. $\lfloor \frac{\lfloor 16x \rfloor}{16} \rfloor$ is very close to $\lfloor x \rfloor$ but $\lfloor \frac{\lfloor 16ix \rfloor}{16} \rfloor$
is not close to $\lfloor ix \rfloor$.

(i, j, k)	I_{new}[i][j][k] $\equiv$	
	by Algorithm 2	by Algorithm 1
(0, 2, 1)	I_{old}[1][*][*]	I_{old}[2][*][*]
(1, 2, 3)	I_{old}[3][*][*]	I_{old}[4][*][*]
⋮	⋮	⋮

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.

Note also the presence of floating point multiplications in the above preprocessing $O(N)$ code. It is possible to perform $O(1)$ floating point multiplications (thereby, practically eliminating any floating point calculations) by storing, for instance, the integer part of $D \times xx[1]$. In this case, the errors will be considerably higher.

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 is 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 $D = 10000$ 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 on to 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.e., iv or transform_index[i][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 [GLS88]. 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 $\frac{p_x}{q}, \frac{p_y}{q}, \frac{p_z}{q}$ with the same denominator q, such that for all i, j, k in the interval $-N, \ldots, N$,

$$trunc(ir_1 + jr_2 + kr_3) = trunc(i\frac{p_x}{q} + j\frac{p_y}{q} + k\frac{p_z}{q}) \quad (1)$$

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 $0 < \epsilon < 1$, computes SDA approximation integers $p_1, p_2, \ldots, p_n$, and an integer q such that

$$1 \leq q \leq 2^{n(n+1)/4}\epsilon^{-n}$$
$$\text{and} \quad |\alpha_i q - p_i| < \epsilon \quad (i = 1, \ldots, n)$$

Proof : See [GLS88].

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, $N \in \mathbb{Z}$, $0 < \epsilon < \frac{1}{3N}$ and $\frac{p_1}{q}, \frac{p_2}{q}$ and $\frac{p_3}{q}$ be their SDA approximations, then $\nexists z \in \mathbb{Z}$ such that

$$ir_1 + jr_2 + kr_3 < z \leq i\frac{p_1}{q} + j\frac{p_2}{q} + k\frac{p_3}{q}, \quad i, j, k \leq N \quad (2)$$

or

$$i\frac{p_1}{q} + j\frac{p_2}{q} + k\frac{p_3}{q} < z \leq ir_1 + jr_2 + kr_3, \quad i, j, k \leq N \quad (3)$$

Proof : Since $\frac{p_1}{q}, \frac{p_2}{q}$ and $\frac{p_3}{q}$ are the Simultaneous Diophantine Approximations of r_1, r_2 and r_3 we have, from Theorem 1,

$$|qr_i - p_i| < \epsilon, \qquad i = 1, 2, 3. \quad (4)$$

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

$$\therefore \quad i\frac{p_1}{q} + j\frac{p_2}{q} + k\frac{p_3}{q} < z \leq ir_1 + jr_2 + kr_3$$
$$\text{or,} \quad ip_1 + jp_2 + kp_3 < zq \leq iqr_1 + jqr_2 + kqr_3$$

Subtracting, we get

$$0 < zq - ip_1 - jp_2 - kp_3 \leq i(qr_1 - p_1) + j(qr_2 - p_2) + k(qr_3 - p_3)$$
$$\text{Eqn (4)} \Rightarrow \; 0 < zq - ip_1 - jp_2 - kp_3 < i\epsilon + j\epsilon + k\epsilon$$
$$0 < zq - ip_1 - jp_2 - kp_3 < 3N\epsilon$$
$$0 < zq - ip_1 - jp_2 - kp_3 < 1$$

resulting in a contradiction that there exists an integer between 0 and 1. Hence $\nexists z \in \mathbb{Z}$ satisfying Equation 3.

A similar argument implies that $\nexists z \in \mathbb{Z}$ satisfying Equation 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 4×4 of floating point numbers. Having obtained the rational simultaneous approximations $\frac{p_{xx}[a]}{q[a]}, \frac{p_{yy}[a]}{q[a]}, \frac{p_{zz}[a]}{q[a]}$, $a = 1, 2, 3$, the computation of $iv[a]$ is accomplished completely in integer arithmetic as follows.

We may optionally pre-compute and store D_{xx}, D_{yy} and D_{zz} as described in Algorithm 3.

Now by Theorem 2, the vector iv obtained by:

iv[1] := (D_{xx}[i][1] + D_{yy}[j][1] + D_{zz}[k][1]) *div* q[1];
iv[2] := (D_{xx}[i][2] + D_{yy}[j][2] + D_{zz}[k][2]) *div* q[2];
iv[3] := (D_{xx}[i][3] + D_{yy}[j][3] + D_{zz}[k][3]) *div* q[3];

matches, in the sense of Equation 1 the ideal vector transform_index[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 $9N$.

Algorithm 3 Proposed pseudo code

```
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}
for loopVar := -N to N do
    $D_{xx}$[loopVar][1] := loopVar * $p_{xx}$[1];
    $D_{xx}$[loopVar][2] := loopVar * $p_{xx}$[2];
    $D_{xx}$[loopVar][3] := loopVar * $p_{xx}$[3];
    $D_{yy}$[loopVar][1] := loopVar * $p_{yy}$[1];
    $D_{yy}$[loopVar][2] := loopVar * $p_{yy}$[2];
    $D_{yy}$[loopVar][3] := loopVar * $p_{yy}$[3];
    $D_{zz}$[loopVar][1] := loopVar * $p_{zz}$[1];
    $D_{zz}$[loopVar][2] := loopVar * $p_{zz}$[2];
    $D_{zz}$[loopVar][3] := loopVar * $p_{zz}$[3];
end for
```

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 transform_index[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):

transform_index[i][j][k][a] = (D_{xx}[i][a] + D_{yy}[j][a] + D_{zz}[k][a]) div q[a];
transform_index[i][j][k][a] =
transform_index[i][j][k - 1][a] + p_{zz}[a] div q[a];

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 $\frac{p_{zz}[a]}{q[a]}$ 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 memoization 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

```
Procedure mapVoxels ($p_{xx}, p_{yy}, p_{zz}$, q : integer; comp : 1 .. 3)
    var residue : array[0 .. N][0 .. N] of integer;
begin
    transform_index[0][0][0][comp] := 0;
    residue[0][0] := 0;
    for i := 1 to N do
        residue[i][0] := residue[i - 1][0] + $p_{xx}$;
        if residue[i][0] $<$ q then
            transform_index[i][0][0][comp] :=
            transform_index[i - 1][0][0][comp];
        else
            transform_index[i][0][0][comp] :=
            transform_index[i - 1][0][0][comp] + 1;
            residue[i][0] := residue[i][0] - q;
        end if
    end for
    for i := 0 to N do
        for j := 1 to N do
            residue[i][j] := residue[i][j - 1] + $p_{yy}$;
            if residue[i][j] $<$ q then
                transform_index[i][j][0][comp] :=
                transform_index[i][j - 1][0][comp];
            else
                transform_index[i][j][0][comp] :=
                transform_index[i][j - 1][0][comp] + 1;
                residue[i][j] := residue[i][j] - q;
            end if
        end for
    end for
    for i := 0 to N do
        for j := 0 to N do
            for k := 1 to N do
                residue[i][j] := residue[i][j] + $p_{zz}$;
                if residue[i][j] $<$ q then
                    transform_index[i][j][k][comp] :=
                    transform_index[i][j][k - 1][comp];
                else
                    transform_index[i][j][k][comp] :=
                    transform_index[i][j][k - 1][comp] + 1;
                    residue[i][j] := residue[i][j] - q;
                end if
            end for
        end for
    end for
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).

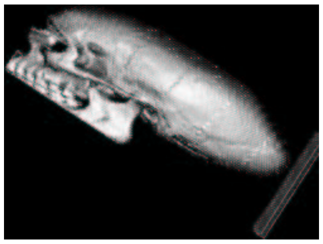
Fig. 4. Result obtain using naive integer approximations (Algorithm 2)

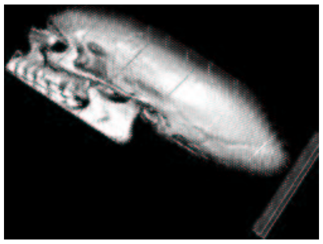
Fig. 5. Results obtained using Algorithm 4 is visibly different.

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 three dimensional 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, Figures 4 and 5 show the results obtained from using Algorithms 2 (a traditional method) and 4 (our proposed method) to rotate the 3D image of a skull by angles 1.0, 1.0 and 33 degrees about the 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 [FC97] 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 are 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 $9N$ 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 A: Solving simultaneous Diophantine equations

The algorithm to solve SDA ([GLS88]) is shown in Algorithm 5; an implementation of this will satisfy Equation 1.

Algorithm 5 Approximate with error less than epsilon

```
1:  Procedure SDApproximate ($\alpha$[1..n], $\epsilon$ : real; p[1..n], q : integer)
2:    $B := \begin{bmatrix} I_{n\times n} & \alpha \\ 0_{1\times n} & 2^{-n(n+1)/4}\epsilon^{n+1} \end{bmatrix}$;
3:    $B^*$ := GramSchmidtOrthogonalize($B$);
4:    loop
5:      let $\mu := BB^{*T}$ $\{B^*$ is orthogonal$\}$
6:      for j := 1 to n do
7:        for i := 1 to j - 1 do
8:          $B_j := B_j$ - round $(\mu_{ji})B_i$;
9:        end for
10:     end for
11:     if $\exists j$ such that $\|B^*_{j+1} + \mu_{j+1,j}B^*_j\|^2 < \frac{3}{4}\|B^*_j\|^2$ then
12:       swap $B_j$ and $B_{j+1}$;
13:       adjust $B^*$ accordingly;
14:     else
15:       break;
16:     end if
17:     $q := 2^{n(n+1)/4}\epsilon^{-(n+1)}B_{1,n+1}$
18:     $p := B_1 + q\alpha$
19:   end loop
20: end Procedure
```

References

[BB82] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.

[CT99] R. Cox and R. Tong. Two- and three-dimensional image rotation using the fft. *IEEE Transactions on Image Processing*, 8:1297–1299, 1999.

[FC97] M. Fleury and A. Clark. Sampling concerns in scanline algorithms. *IEEE Transactions on Medical Imaging*, 16:349–361, 1997.

[GLS88] Martin Grotschel, Laszlo Lovasz, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag Berlin Heidelberg, 1988.

[GM95] I. Ghosh and B. Majumdar. Vlsi implemtnation of an efficient asic architecture for real time rotation of digital images. *International Journal of Pattern Recognition and Artificial Intelligence*, 9:449–462, 1995.

[Hor86] B. K. P. Horn. *Robot Vision*. MIT Press, Cambridge, 1986.

[Nel93] Mark Nelson. *The Data Compression Book*. M & T Publishing, Inc., 1993.

[Pra80] J. M. Prager. Extracting and labeling boundary segments in natural scenes. *IEEE Transactions in Pattern Analysis and Machine Intelligence*, 2(1):16–27, January 1980.

Processor	Time to execute user instructions			% gain of 4 over 1
	Algorithm 1	Algorithm 2	Algorithm 4	
Pentium 100	32.34s	14.65s	11.78s	174.53%
Pentium 60	48.71s	57.30s	22.90s	112.70%
Alpha 233	26.33s	53.22s	19.56s	34.61%
SGI R5000	29.13s	28.36s	9.88s	194.83%
Sun 4m	24.79s	10.63s	15.06s	64.60%
Intel 386*	20.99s	4.80s	4.58s	358.29%

$^*N = 64$

TABLE II

COMPARISON OF THE PERFORMANCE OF THE ALGORITHMS ON DIFFERENT MACHINES.

[TQ97] Tommaso Toffoli and Jason Quick. Three-dimensional rotations by three shears. *Graphical models and image processing: GMIP*, 59(2):89–95, 1997.

[Vol59] J. Volder. The CORDIC trignometric computing technique. *IRE Transactions on Electronics and Computing*, EC-8:330–334, 1959.