Application of doo subdivision for planning roughing cuts in five axis machining

Abhishek Saboo

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation

https://lib.dr.iastate.edu/rtd/20262

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Application of doo subdivision for planning roughing cuts in five axis machining

by

Abhishek Saboo

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Industrial Engineering

Program of Study Committee:
Ranga Narayanaswami, Major Professor
Carolyn Heising
Palaniappa Molian

Iowa State University
Ames, Iowa
2004
Graduate College
Iowa State University

This is to certify that the masters thesis of

Abhishek Saboo

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Three Axis NC Machine</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1. Point-to-Point positioning control</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2. Straight cut positioning control</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3. Contouring positioning control</td>
<td>4</td>
</tr>
<tr>
<td>1.2. Five Axis NC Machining</td>
<td>5</td>
</tr>
<tr>
<td>CHAPTER 2. WAVELETS AND MULTiresOLUTION ANALYSIS</td>
<td>9</td>
</tr>
<tr>
<td>2.1. Chaikin’s Subdivision</td>
<td>12</td>
</tr>
<tr>
<td>2.2. Doo Subdivision</td>
<td>13</td>
</tr>
<tr>
<td>CHAPTER 3. APPLICATION OF DOO SUBDIVISION IN COMPUTER AIDED DESIGN AND MANUFACTURING</td>
<td>18</td>
</tr>
<tr>
<td>3.1. Getting the shape of object using Doo Subdivision</td>
<td>19</td>
</tr>
<tr>
<td>3.2. Determining the sequence of operation using mat lab.</td>
<td>21</td>
</tr>
<tr>
<td>3.3. Machining time simulation.</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 4. RESULTS AND CONCLUSION</td>
<td>26</td>
</tr>
<tr>
<td>CHAPTER 5. FUTURE WORK</td>
<td>28</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>32</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>41</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to sincerely thank, Dr. Ranga Narayanaswami, for his invaluable guidance. I am grateful to him for encouraging me and providing me support throughout.

I would also like to thank my committee members Dr. Carolyn Heising and Dr. Molian for being on my committee and give me much appreciated guidance.

Finally, I would like to thank my parents and all my friends for helping me and giving moral support.
Wavelets permit multiresolution analysis of curves and surfaces. A complex curve can be decomposed using wavelet theory into lower resolution curves. The main idea of Subdivision is “Subdivision defines a smooth surface as a limit of a sequence of successive refinements”. Subdivision methods are quite useful for generating surface in computer graphics. A fine approximation of the model is obtained by successfully subdividing the vertices of the coarse approximation.

In this research we have worked on a novel idea of application of Doo subdivision in machining of surfaces using Multi axis Machine. Here we have first generated subdivision surfaces using Doo subdivision followed by the sequence of operations required for roughing.

We have data, which suggests using this approach on multi axis machine; reduce machining time in comparison with three axis NC machine. Along with that will also generate better quality-roughed surface, which will help to reduce machining time for finishing too.
CHAPTER 1. INTRODUCTION

One of the important aspects of Machining is reducing machine time and improving the quality. In the rough-cut stage, the main goal is to remove material in the most efficient manner. In finish cutting, producing the desired surface finish and accuracy is the primary driving factor. Till recently the industry relied a lot on 3-axis machine for machining of surfaces. Significant amount of research and development has been done in the field, which has led to development of better NC programming methods and machining tools. Even then machining them on 3-axis machine using special tools and multiple setups compromises the practical limits of accuracy and cost for complex parts. The main Objective of this research is to develop a process to reduce roughing time, improving the surface quality using five axis machines.

1.1) Three axis Machine

Three axis mills are available in horizontal and vertical spindles configurations. A vertical spindle machine is the choice of die-sinking and other operation requiring work on one face of the part only. Fixturing is usually rudimentary in vertical machine spindle work, since the part can be clamped to the worktable or held in suitable milling vise. Chip accumulation, however, is an inconvenience.

For production work, or work requiring operations on more than one face of the part, as is necessary in the manufacture of fluid power pumps, valves, and manifolds, am horizontal
spindle is preferred. In this chip accumulation is a lesser problem. Both horizontal and vertical spindle machine are available as positioning or contouring machines. The horizontal spindle machine configuration is well adapted to receive additional equipment to extend its versatility. Automatic tool changers with tool magazine, rotating worktables may be added.

There are three types of positioning control in Three-axis machine:

When classified according to the machine tool control system, there are three basic types of NC systems:

1.1.1. Point to Point.
1.1.2. Straight cut.
1.1.3. Contouring.

The classification is concerned with the amount of control over the relative motion between the work piece and cutting tool. The least control is exerted over the tool motion with the point-to-point systems. Contouring represents the highest level of control.

1.1.1) Point-to-Point NC positioning control

The principal function of the point-to-point positioning control is to position the tool from one point to another within a coordinate system; therefore the control is most often referred to as point-to-point NC system. In PTP the objective of the machine tool control system is to move the cutting tool to predefined location. The speed or path by which this movement is accomplished is not important in point to point NC. Once the tool reaches the desired location, the machining operation is performed at that position. NC drill presses are a good example of PTP systems.
The spindle must first be positioned at a particular location at the work piece. This is done under PTP control. Then the drilling of the holes is performed at that location, the tool is moved to the next whole location, and so forth. Since no cutting is performed between holes there is no need for controlling the relative motion of the tool and work piece between hole locations. On positioning systems the speeds and feeds used by the machine tool are often used by the machine operator rather than by the nc tape. Positioning systems are the simplest machine tool control systems and therefore the least expensive of the three types. However for certain operations such as drilling and spot welding, point-to-point is perfectly suited to task and any higher level of control is unnecessary.

1.1.2) Straight Cut NC positioning control

Straight cut positioning systems provide a limited degree of control during the positioning of the tool from one point to other. Most of the straight cut systems are fitted with manually adjusted feed control. All the programmable axes of the NC machine, which allows the system to perform milling, in addition to drilling, share this feed control. Straight cut control systems are capable of moving the cutting tool parallel to one of the major axes at a controlled rate suitable for machining. It is therefore appropriate for performing milling operations to fabricate work pieces of rectangular configurations. With this type of NC systems it is therefore appropriate for performing milling operations to fabricate work pieces of rectangular configurations. With this type of NC system it is not possible to combine movements in more than single axis direction. Therefore angular cuts on the work piece would not be possible. An NC machine tool capable of performing straight cut movements is also capable of point-to-point movements.
1.1.3) Contouring NC positioning control

The contouring system is the state of art, high technology most versatile and intricate of the CNC devices. It generates a continuously controlled tool path, by interpolating intermediate points or coordinates. Interpolating here means the capability of computing the points of the path. Contouring is the most complex flexible and the most expensive type of machine tool control. It is capable of performing both PTP and straight cut operations. In addition the distinguishing feature of the of contouring NC system is their capacity for simultaneous control of more than one axis movement of machine tool. Figure 1.1 below illustrate the versatility of continuous path NC. Milling and Turning are the common examples of the use of contouring control.

Figure 1.1 Continuous path NC
1.2) Five Axis NC Machining

The need for reducing machining time, production cost and improving machining quality led to the development of five axis machines. To achieve the intended performance of a product (e.g., an impeller or turbine blade) or capture its desired aesthetics (e.g., a car fender or hood), engineers and designers continually seek new methods and techniques for specifying part geometry. This has resulted in the development and use of entirely new classes of surfaces.

Traditionally, the tool paths required to machine complex surfaces have been constructed using numerical control languages or CAD systems. These languages or systems generate numerical instructions for multi-axis machine tools based on defined part geometry and specified machining parameters. The numerical instructions are generated by first reducing the dimensionality of the surface to that of planar or isoparametric curves.

Milling machines used to machine complex surfaces have three axes of simultaneous tool positioning control and zero, one or two axes of simultaneous tool orientation control. While 3-axis milling machines (those with no tool orientation capabilities) have historically been the choice to machine complex surfaced parts, five axis-milling machines have increasingly been used in industry. As compared to 3-axis machining, five axis machining (a simple example is shown in Figure 1.2) offers many advantages such as higher productivity and better machining quality. In five axis machining, the orientation of the tool can be determined by the two additional degrees of freedom so as to obtain efficient tool paths. Figure 1.3 shows the tool orientation control in five axis machining. The angle of the tool in the plane of
motion is called the tool inclination angle $\alpha$, and the angle of tool out of the plane of motion is called the tool tilt angle $\beta$.

Figure 1.2 Five axis machining example

Figure 1.3 Tool orientation in five axis machining.
Wavelets can provide a single theoretical framework for performing most of the geometric reasoning needed for efficient tool path generation in five axis NC machining. The application of wavelets can also minimize the need for these geometric algorithms and improve the computational efficiency when these geometric algorithms are needed. Most important, wavelet based multiresolution representations generate a series of intermediate shape models as the references of NC programming, in which the tool cutting ability and removal volume can be automatically considered, the amount of model data can be compressed, and the resolution of the models can be changed. In addition, multiresolution models represented by polygonal meshed surfaces are becoming more popular in CAD/CAM. However, there is almost no published literature about tool path generation directly from a multiresolution meshed representation. The wavelet representation is very conducive for tool path generation based on a meshed surface.

Wavelets and multiresolution analysis may be used to hierarchically represent the original surface. Each hierarchical model successively approximates the final object shape. Thus rough-cuts can be made with the low-resolution models and finish cuts can be made with the high-resolution models. Wavelets may also be used to develop new multiresolution accessibility analysis algorithms for efficient collision detection and avoidance.

In this research we demonstrate the application of Doo Subdivision based on the theory of wavelets in machining. This is a novel application and provides added insight and a rich theoretical framework for some new process for machining. The appearance of wavelets is a relatively recent development in mathematics. They give rise to hierarchical representation
and have been successfully used for various tasks such as approximation theory, signal processing, image storage and compression, and content based retrieval. Recently the theory of wavelets and multiresolution analysis has also been applied to hierarchical editing of curves and surfaces in computer graphics.

The main idea of Subdivision is "Subdivision defines a smooth surface as a limit of a sequence of successive refinements". Subdivision methods are quite useful for generating surface in computer graphics. A fine approximation of the model is obtained by successfully subdividing the vertices of the coarse approximation.

The rest of the report is organized as follows. The theory of multiresolution analysis and wavelets is briefly reviewed along with Chaikens and Doo Subdivision in chapter 2. In chapter 3, how did we approached to this problem has been discussed. It basically involves getting the shape of object using Doo Subdivision, Determining the sequence of operation; machining time simulation. In chapter 4, Results along with conclusion is presented, followed is the proposed future work along with references and appendix.
CHAPTER 2. WAVELETS AND MULTiresolution analysis

This chapter provides some background on wavelets and multiresolution analysis. Multiresolution analysis is a simple mathematical tool that has found a wide variety of applications in recent years, including signal analysis (Mallet, 1989), image processing (DeVore et al., 1992), and numerical analysis (Beylkin et al., 1991). Informally, wavelets are the basis functions for multiresolution analysis.

A signal or a function may be better understood if expressed as a linear decomposition over a basis.

\[ f(t) = \sum_i a_i \phi_i(t) \]  

(2.1)

The basis functions are usually chosen to be orthogonal. For the Fourier series these basis functions are \( \sin(k\omega_0 t) \) and \( \cos(k\omega_0 t) \). On the other hand, the wavelet expansion is a two-parameter system described by

\[ f(t) = \sum_k \sum_j a_{j,k} \psi_{j,k}(t) \]  

(2.2)

The coefficients \( a_{j,k} \) are called as the discrete wavelet transform. The wavelet expansion gives a time frequency localization of the signal. A wavelet representation is much like a musical score where the location of the notes tells when the tones occur and what their frequencies are. Wavelet systems are generated from a single scaling function by simple scaling and translation. Therefore, if a set of functions is represented by a weighted-sum of \( \psi(t - k) \) then a larger set of functions may be represented by \( \psi(2t - k) \). Wavelets thus satisfy
the multiresolution conditions. The lower resolution coefficients can be calculated from the higher resolution coefficients, using a tree-structured algorithm called a *filter bank*.

Consider a discrete signal $C^n$, expressed as a column vector of samples $[c^n_1, ..., c^n_m]^T$. In our curve application described in Chapter 3, for example, the samples $c^n_i$ will be the curve’s control points in $\mathbb{R}^2$.

Suppose we wish to create a low-resolution version $C^{n-1}$ of $C^n$ with a fewer number of samples $m'$, the standard approach for creating the $m'$ samples of $C^{n-1}$ is to use some form of linear filtering and sub-sampling on the $m$ samples of $C^n$. This process can be expressed as a matrix equation

$$C^{n-1} = A^n C^n$$

(2.3)

Where $A^n$ is an $m' \times m$ matrix.

Since $C^{n-1}$ contains fewer samples than $C^n$, it is intuitively clear that some amount of details is lost in this filtering process. If $A^n$ is chosen appropriately, it is possible to capture the lost detail as another signal $D^{n-1}$ with $m - m'$ samples, computed by

$$D^{n-1} = B^n C^n$$

(2.4)

Where $B^n$ is an $(m - m') \times m$ matrix, which is related to matrix $A^n$. The pair of matrices $A^n$ and $B^n$ are called *analysis filters*. The process of splitting a signal $C^n$ into a low-resolution
version $C^{n-1}$ and detail $D^{n-1}$ is called decomposition. Usually $m$ is roughly half of $m$, so that $C^{n-1}$ and $D^{n-1}$ are roughly equal in size.

Note that $C^{n-1}$ and $D^{n-1}$ together have the same amount of information as $C^n$. If $A^n$ and $B^n$ are chosen correctly, then the original signal $C^n$ can be recovered from $C^{n-1}$ and $D^{n-1}$ by using another pair of matrices $P^n$ and $Q^n$, called synthesis filters, as follows:

$$C^n = P^nC^{n-1} + Q^nD^{n-1}$$ (2.5)

Recovering $C^n$ from $C^{n-1}$ and $D^{n-1}$ is called reconstruction.

The procedure for splitting $C^n$ into a low-resolution part $C^{n-1}$ and a detail part $D^{n-1}$ can be applied recursively to the new signal $C^{n-1}$. Thus, the original signal can be expressed as a hierarchy of low-resolution signals $C^0, \ldots, C^{n-1}$ and details $D^0, \ldots, D^{n-1}$, as shown in Figure 2.1. This recursive process is known as a filter bank.

![Figure 2.1. The filter bank](image)
Since the original signal $C^n$ can be recovered from the sequence $C^0, D^0, D^1, \ldots, D^{n-1}$, this sequence can be thought of as a transform of the original signal known as a \emph{wavelet transform}. Note that the total size of the transform $C^0, D^0, \ldots, D^{n-1}$ is the same as that of the original signal $C^n$, so no extra storage is required.

### 2.1) Chaikin's Subdivision

Since Doo subdivision may be explained as Chaikin’s subdivision rule for curves extended to surfaces, here first explained is geometrically how to determine the Chaikin local reverse subdivision rule. Chaikin’s subdivision is a “corner-cutting” strategy. Figure 2.2 shows one step of this subdivision as a two-step process (for simplicity, we use $vi$ to stand for $vi_k$ and $wj$ for $vj_k$).

Two new points are first determined on every line segment, and then the second new point on each successive line segment is connected with the first new point of the next segment. It can be shown that the points generated by repeatedly applying Chaikin’s subdivision rule

![Figure 2.2 A two stage representation of Chaikin’s Rule](image)
2.2) Doo Subdivision

Doo subdivision is a “corner-cutting” method, for the representation of three-dimensional surfaces. In this method, the shapes of the faces of the surface are assumed to be arbitrary (consequently, if a face is not triangular, it may not be planar). It can be shown that the Doo subdivision scheme is an extension of Chaikin’s method. To show this, consider Chaikin’s method in two stages as shown in Figure 2.2.

In the first stage, contract each line segment of the polygon towards its center by a factor \( \alpha \). For example in Figure 2.2, \((v_0, v_1)\) is contracted into \((w_0, w_1)\), where \( w_0 = \alpha d + (1-\alpha)v_0 \), \( d = (v_0 + v_1)/2 \), and so on for the other points. In the second stage for those line segments having been joined together before the subdivision, introduce a new line segment by joining the corresponding end points of their contracted line segments. Again in figure 2.2, the line segment \((w_0, w_1)\) is joined with \((w_2, w_3)\) through the new line segment \((w_0, w_1)\). The multiplying factor \( \alpha \) should be chosen such that the limiting curve should be \( C^1 \) curve.

For the Doo Subdivision of surfaces, the operations above on the line segment of a polygon are extended to the faces of the polyhedron: the faces are contracted towards their centroids and the contracted versions of adjoining faces are joined by introducing additional new faces for example in the figure 2.3.
Figure 2.3 A simple doo subdivision

The original surface is composed of the two adjoining faces $f_1$ and $f_2$. In the next stage, $g_1$ and $g_2$ are joined by $g_3$ to observe the effect on a three dimensional figure (cube), where Doo subdivision is applied on a cube.

To explain more completely, assume that $(F^k, V^k)$ is a surface of arbitrary topological type, being represented by Faces $F^k$ and vertices $V^k$. Each vertex $v \in V^k$ is identified by its coordinates in space and each face by the sequence of its associated vertices. Let $E$ be the set of existing edges $(F^k, V^k)$. Now is the explanation, how to identify a subdivided surface $(F^{k+1}, V^{k+1})$ as a result of application of Doo subdivision scheme on $(F^k, V^k)$. First, for each $f_i \in F^k$ introduce a new face $g_i \in F^{k+1}$ by contraction, as in figure 2.3 in the following way: If $f_i=(v_1, \ldots, v_n)$ is the faces in original faces and $g_i=(w_1, \ldots, w_n)$ is the contracted faces, the $w_\mu$ come from contracting $v_\lambda$. With respect to the centroid of $f_i$ as follows (for any reasonable, systematic indexing system for the $f, g, v,$ and $w$):

$$w_\mu = (1-\alpha)d + \alpha v_\lambda.$$  
$$d = 1/n(v_1 + v_2 + \ldots + v_n)$$
In the above equation the value of $\alpha$, $0 < \alpha < 1$, can be chosen so that the limiting surface will be $C^1$. After this operation, the set of $w_{\mu}$ obtained by the contraction of all faces, forms the new set $V^{K+1}$. Note that each $v \in V^k$ corresponds to $\deg(v)$ (the degree of $v$: that is, the number of faces joined at $v$) new vertices introduced in $V^{k+1}$ (Fig 2.4).

The set of new faces in $F^{K+1}$ is composed of the following three types.

1) Face-to-face: The new set of faces, $g_i$, is obtained from $F^k$, denoted by $F_F$.

2) Face-to-edge: If $f_i$ is joined with $f_j$ through an edge $e$ in the graph $(F^k, V^k)$, then we have face $g_e$ joining $g_i$ and $g_j$. Therefore for each edge not lying on the boundary, a new face is introduced denoted by $F_E$. Each face in $F_E$ is rectangular.

3) Face-to-vertex: For each internal vertex $v$ in $(F^k, V^k)$ a new face $g_v$ is introduced, so that it joins the corresponding new vertices $w_j$ of $v$. Denoted by $F_v$. Number of sides for each force in $F_v$ is equal to the degree of its corresponding vertex in $V^k$. 
Figure 2.4 Conversion of $v$ to $w_\phi, w_\pi, w_\gamma, w_\delta$
It can be easily verified that

- Each face in $F_F$ has the same number of edges as the face from which it is contracted.
- Each face in $F_E$ is rectangular and
- The number of sides for each face in $F_v$ is equal to the degree of its corresponding vertex in $V^k$.

Determining these three types of faces, the subdivision is complete. Therefore we have

$$F^{k+1} = F_F + F_E + F_v.$$
CHAPTER 3. APPLICATION OF DOO SUBDIVISION IN COMPUTER AIDED DESIGN AND MANUFACTURING

Lot of work has been done till date using idea of subdivision in curves and surfaces in the field of computer graphics, here we are trying to develop an innovative idea of application of subdivision in Manufacturing. In this section application of Doo subdivision in computer aided design and manufacturing has been explained. It's a new idea so no literature was available for this.

The main Idea of Subdivision is “Subdivision defines a smooth surface as a limit of a sequence of successive refinements”. We are using the idea of subdivision such that, we start from initial stock and then by using doo subdivision once, we get the shape we should have using multi axis machine and similar repetitions leads us to the final product.

Subdivision methods are quite useful for generating surface in computer graphics. A fine approximation of the model is obtained by successfully subdividing the vertices of the coarse approximation.

In this research we have approached this application basically in three steps:

3.1) Getting the shape of object using Doo Subdivision

3.2) Determining the sequence of operation using mat lab.

3.3) Machining time simulation.
3.1) **Getting the shape of object using Doo Subdivision:**

In this step we apply doo subdivision and get the shape of the figure we are going to manufacture. Working on mat lab does this; basically we apply doo subdivision to look the shape of the figure. Here we have some figures, which we get after applying Doo Subdivision. Initially some coordinate points have been given for the cube as input parameters. The codes used to get these shapes have been attached in the appendix.

Followed are the different surfaces after application of Doo Subdivision:

![Figure 3.1 Doo subdivision on cube](image)
Figure 3.2 Doo subdivision on menger

Figure 3.3 Doo subdivision on tetra
3.2) Determining the sequence of operation using mat lab:

Doo Subdivision leads to different steps, after applying subdivision for the first time the result lead us to the surface which is needed after first roughing operations. Here for example in the case for convex figure like cube we start with a cube and after first roughing operation using multi axis machine we want to have it same as cube will look after application of doo subdivision once. So here we can see basically doo subdivision is leading us to the shape, which we want after each roughing. But in order to determine the right sequence to get the desired shape we have generated Code to look at shape of the original cube exactly after each cut. The main motive behind this is to give rough cuts in a sequence of operations to get the required convex figure with least number of set ups. Similarly second roughing operation is
done to get the following figure. We have done this work by analyzing movement of cutting plane and original plane in mat lab, which can be seen in the figure (3.5 and 3.6)

Figure 3.5 Sequence of operations

Figure 3.6 Step I after machining
Here is the code used for generating the step 1 and by using this code with different plot commands one can see how the figure look as shown in fig.

\[
x = [0 1 1 0 0 0 ; 1 1 0 0 1 1 ; 1 1 0 0 1 1 ; 0 1 1 0 0 0 ; 0 1 1 0 0 0 ; 0 0 0 1 0 0 ; 1 1 1 0 1 1 ; 1 1 1 0 1 1 ; 0 0 0 0 0 1 0 ; 1 1 1 0 1 1 ; 1 1 1 0 1 1 ; 0 0 0 0 0 1 0 ] ;
\]
\[
y = [0 0 1 1 0 0 0 ; 0 1 1 0 0 0 0 ; 0 1 1 0 1 1 0 ; 0 0 1 1 1 0 0 ; 0 0 1 1 0 0 0 ; 0 0 0 1 0 0 ; 1 1 1 0 1 1 ; 1 1 1 0 1 1 ; 0 0 0 0 0 1 0 ; 0 0 0 0 0 1 0 ; 0 0 0 0 0 1 0 ; 0 0 0 0 0 1 0 ] ;
\]
\[
z = [ 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0 ] ;
\]

\[
\begin{align*}
&\text{for } i = 1:1:7 \\
&\quad \text{temp}(1) = 0.0; \\
&\quad \text{temp}(2) = 0.0; \\
&\quad \text{temp}(3) = 0.0; \\
&\quad \text{for } j = 1:1:5 \\
&\quad \quad \text{temp}(1) = (\text{temp}(1) + x(j,i)); \\
&\quad \quad \text{temp}(2) = (\text{temp}(2) + y(j,i)); \\
&\quad \quad \text{temp}(3) = (\text{temp}(3) + z(j,i)); \\
&\quad \end{align*}
\]

\[
\begin{align*}
&\quad d(i,1) = \text{temp}(1)/5; \\
&\quad d(i,2) = \text{temp}(2)/5; \\
&\quad d(i,3) = \text{temp}(3)/5; \\
&\end{align*}
\]

\[
\begin{align*}
&\text{for } i = 1:7 \\
&\quad \text{for } j = 1:5 \\
&\quad \quad x(j,i) = d(i,1)/2 + x(j,i)/2; \\
&\quad \quad y(j,i) = d(i,2)/2 + y(j,i)/2; \\
&\quad \quad z(j,i) = d(i,3)/2 + z(j,i)/2; \\
&\end{align*}
\]

\[
\begin{align*}
&X = [x(2,1), x(2,5), x(1,2), x(2,1)]; \\
&Y = [y(2,1), y(2,5), y(1,2), y(2,1)]; \\
&Z = [z(2,1), z(2,5), z(1,2), z(2,1)]; \\
&X1 = [x(4,5), x(2,3), x(1,4), x(4,5)]; \\
&Y1 = [y(4,5), y(2,3), y(1,4), y(4,5)]; \\
&Z1 = [z(4,5), z(2,3), z(1,4), z(4,5)]; \\
&X2 = [x(1,5), x(2,4), x(1,1), x(1,5)]; \\
&Y2 = [y(1,5), y(2,4), y(1,1), y(1,5)]; \\
&Z2 = [z(1,5), z(2,4), z(1,1), z(1,5)]; \\
&X3 = [x(3,2), x(3,6), x(4,3), x(3,2)]; \\
&Y3 = [y(3,2), y(3,6), y(4,3), y(3,2)]; \\
&Z3 = [z(3,2), z(3,6), z(4,3), z(3,2)]; \\
&X4 = [x(4,2), x(2,6), x(3,1), x(4,2)]; \\
&Y4 = [y(4,2), y(2,6), y(3,1), y(4,2)]; \\
&Z4 = [z(4,2), z(2,6), z(3,1), z(4,2)]; \\
&X5 = [x(4,4), x(4,6), x(3,3), x(4,4)]; \\
&Y5 = [y(4,4), y(4,6), y(3,3), y(4,4)]; \\
&Z5 = [z(4,4), z(4,6), z(3,3), z(4,4)]; \\
&X6 = [x(3,4), x(1,6), x(4,1), x(3,4)];
\end{align*}
\]
plot3(x, y, z, X, Y, Z, X1, Y1, Z1, X2, Y2, Z2, X3, Y3, Z3, X4, Y4, Z4, X5, Y5, Z5, X6, Y6, Z6, X7, Y7, Z7)
hold on
plot3(x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4, x5, y5, z5, x6, y6, z6, x7, y7, z7, x8, y8, z8, x9, y9, z9, x10, y10, z10, x11, y11, z11, x12, y12, z12)
Also the shape that we get after following the step method is below.

Figure 3.7 Shape after step procedure

3.3) Machining time simulation
Machining time is a function of the length of the machining segment and the shape complexity of the machining segment for a given feed rate. To calculate the machine time we collected all the data, which are necessary to calculate machining time which includes power of machine, spindle speed, cutter data, feed rate, depth of cut, length of cut etc. After determining all the data required we calculated the machining time required getting to do subdivision step 2.
CHAPTER 4. RESULTS AND CONCLUSION

Doo subdivision along with multi axis machining has given us much better results than using the traditional contouring method used on NC machine. After running simulations for both the methods in case of the cube we got to the following results.

Table 4.1 Comparison of results

<table>
<thead>
<tr>
<th>Machine and method</th>
<th>Spindle speed</th>
<th>Depth of cut</th>
<th>Total time (Seconds)</th>
<th>Improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>5000 rpm</td>
<td>.25 inch</td>
<td>44.04</td>
<td>21.2</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>5000 rpm</td>
<td>.35 inch</td>
<td>39.53</td>
<td>29.27</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>5000 rpm</td>
<td>.43 inch</td>
<td>35.62</td>
<td>36.27</td>
</tr>
<tr>
<td>3 axis machine + step cut approach</td>
<td>5000 rpm</td>
<td>.25 inch</td>
<td>55.89</td>
<td></td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>4000 rpm</td>
<td>.25 inch</td>
<td>54.18</td>
<td>21.64</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>4000 rpm</td>
<td>.35 inch</td>
<td>48.61</td>
<td>29.69</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>4000 rpm</td>
<td>.43 inch</td>
<td>43.81</td>
<td>36.64</td>
</tr>
<tr>
<td>3 axis machine + step cut approach</td>
<td>4000 rpm</td>
<td>.25 inch</td>
<td>69.14</td>
<td></td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>3000 rpm</td>
<td>.25 inch</td>
<td>71.06</td>
<td>22.1</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>3000 rpm</td>
<td>.35 inch</td>
<td>63.76</td>
<td>30.1</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>3000 rpm</td>
<td>.43 inch</td>
<td>57.47</td>
<td>37</td>
</tr>
<tr>
<td>3 axis machine + step cut approach</td>
<td>3000 rpm</td>
<td>.25 inch</td>
<td>91.22</td>
<td></td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>2000 rpm</td>
<td>.25 inch</td>
<td>104.84</td>
<td>22.57</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>2000 rpm</td>
<td>.35 inch</td>
<td>94.05</td>
<td>30.54</td>
</tr>
<tr>
<td>five axis machine + Subdivision approach</td>
<td>2000 rpm</td>
<td>.43 inch</td>
<td>84.73</td>
<td>37.42</td>
</tr>
<tr>
<td>3 axis machine + step cut approach</td>
<td>2000 rpm</td>
<td>.25 inch</td>
<td>135.4</td>
<td></td>
</tr>
</tbody>
</table>
Here we can see that by using Subdivision approach there is an improvement of nearly 20% for the machining time. Also here we have to consider that along with the improvement in machining time there is a significant difference in the quality too as we can see in the figure 3.7 along with figure 3.1, 3.2, 3.3, 3.4 respectively. Here we can also note that by using multi axis machine we can increase spindle speed much higher than the normal NC machine, which will further reducing the machining time. After using Doo Subdivision method for Roughing it is evident by looking at the figures that the time required for finishing will also reduce. So we can conclude that using Doo subdivision method for machining using multi axis machining has not only improved the machining time for rough cuts but also will generate better quality roughed surface which will help to reduce machining time and quality for finishing too.
CHAPTER 5. FUTURE WORK

In this research we have discussed and worked on convex surfaces. Application of wavelets and Subdivision can be extended to concave surfaces too. Multiresolution subdivision surface offset technique is a key for 3D machining. Visibility Maps can determine the visibility or accessibility information at any resolution from lower resolution and wavelet coefficients. The multiresolution analysis based NC tool path planning is expected to be very efficient because of the attractive linear time complexity of wavelets. It can significantly improve the productivity and the accuracy of machined parts. Here in this section we have discussed the possible problems, which are associated with concave surfaces along with recommended approach.

5.1) Wavelet Decomposition

Wavelet based multiresolution analysis will be used to convert the original meshed surface with arbitrary topology to its multiresolution representation (Figure 5.1). Figure 5.1(b) shows the low-resolution representation of the surface in Figure 5.1(a) by implementing a low pass filter A. A high pass filter B obtains the detail part, consisting of wavelet coefficients. The decomposition process, called analysis, further splits Figure 5.1(b) into an even lower resolution version and corresponding wavelet coefficients (Figure 5.1(c)). The filter bank algorithm culminates with the coarsest level representation in Figure 5.1(d), together with wavelet coefficients at each level.
Tool orientation determination

For a simple convex mesh, the normal direction of every small triangular plane in the mesh, intuitively, can be used to decide the tool orientation for five axis machining. For a complex surface of general topology, the task of tool orientation determination is a local search in the neighborhood of the direction first suggested by visibility analysis. The search criterion is to find the point within each region that results in minimum cusp height. For every sample point in the surface, a set of valid tool orientations will be output after this stage.

5.2) Surface Offset

Obtain intermediate surfaces for machining at different levels of detail. The generation of an intermediate surface is a very important component since the accuracy of CNC tool paths is directly related to the accuracy of the intermediate surface. An issue arises when using these simplified surfaces for machining. They are probably intersecting with the original mesh. To avoid gouging in machining, surface offsetting is necessary. Moreover, to apply the
multiresolution surfaces in machining, the surface of a higher resolution model must be covered with that of a lower resolution model in order to prevent over cut.

In this part, an effective and practical offset surface generation method for multiresolution meshed surfaces will be developed. An offset error analysis method based on wavelets will be developed first, in which the multiresolution control method based on the lifting scheme will be used to control offset error to prevent over cut. The value of the wavelet coefficient indicates the difference between a data value that corresponds to the wavelet coefficient and the average of its neighboring data values. Lifting scheme can be used to decide the position of the low-resolution data that are the neighboring points of a higher resolution point corresponding to the wavelet coefficient.

5.3) Multiresolution accessibility analysis

Perform accessibility analysis hierarchically. In this stage, we use the concept of visibility to determine from which directions a point in the offset surface is likely to be accessible to a tool located outside the convex envelop of the object. In our research, visibility information is represented using visibility maps (Chen and Woo, 1992). Visibility cones, which are clusters of visibility directions for points on a work-piece, can be mapped on to the unit sphere to create a visibility map, as shown in Figure 5.4. In Figure 5.4(a), visibility cone for the shaded region is shown; (b) shows the visibility information for the concave region, formed by surfaces A, B and C, is given by the intersection of the individual visibility hemispheres of its neighboring faces.
We will develop visibility maps calculation software for triangular meshed surface. To determine visibility at any resolution, we propose to use visibility information from the lower resolution representations. Wavelet coefficients are used in the procedure. After visibility analysis, the diameter of the tool and the tool holder has to be considered to ensure that the tool can access a required point without any interference.
REFERENCES


Denis Zorin, Peter Schroder, Tony De Rose, Leif Kobbelt, Adi Levin, Wim Sweldens: Course material (Subdivision for modeling and animation) http://www.mrl.nyu.edu/~dzorin/sig00course/ (Retrieved on 02/11/2003)


Kneisl, G. provided Doo Subdivision code for some surfaces, Institute for analysis and scientific computing, Vienna University of Technology, Austria


Loop, C., 1987, Smooth subdivision surface based on triangles, Master's Thesis, Department of Mathematics, University of Utah.


Wood, Z., Desbrun, M., Schroder, P., and Breen, D., 2000, Semi-regular mesh extraction from volumes, Siggraph.
This research has used this doo subdivision code, part of which was provided by Günter Kneisl.

function varargout=qdoO(argpt,argpgon,n,varargin)
% DOO Doo-Sabin subdivision algorithm
% Given a polyhedral mesh defined by points PT and polygons PGON,
% [PTOUT, PGONOUT] = DOO(PT, PGON, N, PARAM1, VALUE1, PARAM2, ...)
% applies the Doo-Sabin subdivision algorithm N times.
% MESH MUST BE CLOSED AND ORIENTABLE.
% For almost all input data (wrt to Lebesgue-measure), the sequence
% of meshes converges to a C^1 surface. (If for example all points
% of a polygon have the same coordinates (a pathological polygon),
% you may end up with a cusp.)
% PT should be an m-by-3 matrix containing the coordinates
% of the points, PGON should be a cell array of polygons, each
% polygon being a point list (of indices to points).
% Parameters:
% ORIENTATION [Positive Negative {None}]
% OUTPUT [Full {Simple}]
% STATS [{On} Off]
% Set ORIENTATION to 'Positive' if each edge of the
% mesh is passed once in each direction, and polygon points
% rotate counterclockwise with respect to the normal vector.
% Set ORIENTATION to 'Negative' if points rotate clockwise.
% If OUTPUT is set to 'Simple' (the default) the output
% has the same form as the inputs PT and PGON.
% If OUTPUT is set to 'Full' PTOUT is an m-by-5 matrix, one
% row per point, the first three entries being the coordinates,
% the fourth the number of a polygon that the point belongs to,
% and the fifth the index of the point in that polygon.
% OUTPGON(i) is a 2-by-nrpts+1 matrix, the first row
% containing the points of which the polygon consists with the
% first point listed again at the end, the second the indices
% of the adjacent polygons.
% If STATS is 'On' some calculating statistics are displayed.
% Example (Dice):
% pt=[0 0 0;1 0 0;1 1 0;0 1 0;... 
% 0 0 1; 1 0 1; 1 1 1;0 1 1];
% pgon={[1 2 3 4] [5 8 7 6] [1 5 6 2]...
% [3 7 8 4] [2 6 7 3] [4 8 5 1]};
% doo(pt,pgon,3);
% See also PGONDISP PCONTRACE ORIENT CHSURT

global pt pctl len pgon pgon len none edge pgon len nrpts sum nrpts new pt new pgon w;

%*******************************************************************************
%*******************************************************************************
%*******************************************************************************
%*******************************************************************************
%*******************************************************************************
%*******************************************************************************

error(nargchk(3,7,nargin));
if mod(nargin,2) == 0
    error('wrong number of input arguments');
end

if size(argpt,2) ~= 3
    pt = argpt';
else
    pt = argpt;
end

if size(pt,2) ~= 3
    error('pt should be a m-by-3 matrix ');
end

pgon = reshape(argpgon, length(argpgon), 1);

orientation = 'none';
output = 'simple';
checks = 'on';
stats = 'on';

orientation = 0;
or = zeros(length(pgon), 1);
or(1) = 1;

for i = 1:2:length(varargin)
    switch lower(varargin{i}(1:2))
    case 'or'
        % orientation
        switch lower(varargin{i+1}(1:2))
        case 'po'
            % positive
            orientation = 1;
        case 'ne'
            % negative
            for j = 1:length(pgon)
                pgoni = pgon(i);
                pgon{i} = pgoni(length(pgoni):-1:1);
            end
            orientation = 1;
        case 'no'
            % none
            orientation = 1;
        end
    end

    case 'none'
        % no orientation
        orientation = 0;
    end
end
otherwise
    error('Bad value for property ''orientation''.');
end

case 'ou' % output
    switch lower(varargin{i+1}(1))
      case 'f' % full
        output='full';
      case 's' % simple
      otherwise
        error('Bad value for property ''output''.');
      end
    end

case 'st' % stats
    switch lower(varargin{i+1}(1:2))
      case 'on'
      case 'of'
        stats='off';
      otherwise
        error('Bad value for property ''stats''.');
      end
    end
  otherwise
    error(['Invalid property : '' varargin{i} ''']);
end
end

%***************************************************************************
% get additional information about input data
%***************************************************************************
%***************************************************************************
%%%%**
tic;
if orientation==0
  t0=clock;
  pgon=pgonorient(pgon); % orients the polygons
  if stats(1:2)=='on'
    ortime=etime(clock,t0);
    fprintf('
 Orientation : %.2fs',ortime);
  end
end
initpgonglobals(pgon);

pgon=order(pgon);
%***** orders the polygon list such that it starts with the extraordinary
%***** pgons (=pgons with n~4 vertices)
pt=fullpointinfo(pt,pgon);
%***** To each point, this function calculates the polygons
Containing it, and the indices of the point in these polygons.

\begin{verbatim}
initpointglobals(pt);

pgon=addfirstpoint(pgon);
%***** Adds the first point of the pgon to the
%***** end of the pgons's pointlist (convenient when referencing)

pgon=fulledgeinfo(pt, pgon);
%***** To each polygon's edge, this function calculates the index of the
%***** Adjacent polygon, and the index of the edge in this polygon.
%***** POLYGONS MUST BE ORIENTED NEGATIVELY

w=costable(pgon);
%***** Calculates the weights for convex-combination (doo-weights)
\end{verbatim}

%******************* first doo step
***************************
%***** It is useful to treat this step separately, because the polygon-
%***** point data looks much simpler after its completion.(e.g. all vertices
%***** connect exactly 4 edges)
%*************************************************************************

[planeidx, vertidx, edgeidx, ptidx]=linearindexing(pt, pgon);
%***** Defines linear indices of both points and pgons of the next doo-
%***** step.
%***** planeidx contains indices of planepolygons, edgeidx->edgepolygons
%***** vertidx->vertpolygons

[newpt, newpgon]=planepolygons(pt, pgon, planeidx, edgeidx);
%***** Derive from the polygons of the previous step by shrinking.

[newpt, newpgon]=vertpolygons(pt, pgon, newpt, newpgon, vertidx, edgeidx, ptidx);
%***** Calculates the polygons that derive from cutting off the vertices.

newpgon=edgepolygons(newpt, newpgon, pgon, edgeidx, vertidx, ptidx);
%***** Calculates the polygons that derive from cutting off the edges.

%************************ Get some information about new data
**************************
redefineglobals;

nrpts=zeros(pgonlen,1);
for i=1:pgonlen
    nrpts(i)=size(pgon{i},2)-1;
end
sumnrpts=sum(nrpts);

%***** This overrides some settings of redefineglobals.
%***** we merely need it after the first doo-step.

%*************************************************************************
if stats(1:2)=='on'
    tnew=toc;
    fprintf('
 DooStep 1 : %.2fs 
',tnew);
end

for i=1:n-1
    qlinearindexing;
    qplanepolygons;
    qvertexpolygons;
    qedgepolygons;
    redefineglobals;
    if stats(1:2)=='on'
        told=tnew;
        tnew=toc;
        fprintf(' DooStep %d : %.2fs 
',i+1,tnew-told);
    end
end
if stats(1:2)=='on'
    tnew=toc;
    fprintf('
 Total time : %.2fs 
',tnew);
    fprintf('
 Polygons of %d
',pgonlen);
    fprintf('
 Points of %d
',ptlen);
end
%******************************************************************** Output argument check
%********************************************************************
%********************************************************************
***

switch nargout
case 0
  pt=pt(:,1:3);
  for i=1:pgonlen
    pgon{i}=pgon{i}(1,1:nrpts(i));
  end
  pgondisp(pt,pgon);
case 2
  if output(1)=='s'  % simple
    pt=pt(:,1:3);
    for i=1:pgonlen
      pgon{i}=pgon{i}(1,1:nrpts(i));
    end
    varargout={pt pgon};
  else  % full
    for i=1:nonedgepgonlen
      pgon{i}=pgon{i}(1:2,:);
    end
    varargout={pt pgon};
  end
otherwise
  error('Wrong number of output arguments');
end

%******************************************************************** Subroutines
%********************************************************************
%********************************************************************
***

function initpgonglobals(pgon);
%***** we initialize our polygon-global
global pgonlen nrpts sumnrpts

pgonlen=length(pgon);
nrpts=zeros(pgonlen,1);
for i=1:pgonlen
  nrpts(i)=size(pgon{i},2);
end
sumnrpts=sum(nrpts);

function redefineglobals
%***** This converts the output of a doo-step to input for the next
global pt ptlen pgon pgonlen nrpts newpt newpgon nonedgepgonlen sumnrpts
% newpt; pgon = newpgon;

nedgepgonlen = pgonlen + ptlen;

% polygons that do not derive from edges

pgonlen = pgonlen + ptlen + sumnrpts/2;

% #planepgons + #vertpgons + #edgepgons

oldptlen = ptlen;

ptlen = sumnrpts;

nrpts = [nrpts; 4*ones(oldptlen + sumnrpts/2, 1)];

% new polygons have exactly 4 vertices

sumnrpts = sumnrpts + 4*(oldptlen + sumnrpts/2);

function newpgon = addfirstpoint(pgon);
% Adds the first point of the pgon to the
% end of the pgon's pointlist (convenient when referencing)

for i = 1:length(pgon)
    pgoni = pgon{i};
    newpgon{i} = [pgoni pgoni(1)];
end

function newpgon = orderpgon
% This function takes the extraordinary pgons to the
% Beginning of the pgonlist.

global pgonlen nrpts

expgonidx = find(nrpts ~= 4);
expgon = pgon(expgonidx);

expgonlen = length(expgon);
orgonlen = pgonlen - expgonlen;

pgon(expgonidx) = [];
newpgon = [expgon ; pgon];
nrpts = [nrpts(expgonidx); 4*ones(orgonlen, 1)];

function newpt = fullpointinfo(pt, pgon)
% To each point, this function calculates the polygons
% Containing it, and the indices of the point in these polygons.

% Data format: coos = 1-by-3 vector of point's coordinates
% pgon = a 2-by-nrpts matrix, the first row is an
% index vector to the polygons containing this
% point,
% the second indicates the position of the point
% in these polygons

global pgonlen nrpts ptlen
newpt = cell(length(pt),1);
for i = 1:length(pt)
    newpt{i}.coos = pt(i,:);
    newpt{i}.pgon = [];
end

for i = 1:pgonlen
    pgoni = pgon{i};
    for j = 1:nrpts(i)
        newpt{pgoni(j)}.pgon = [newpt{pgoni(j)}.pgon newpt{pgoni(j)}.pgon(i,j)];
    end
end

function initpointglobals(pt)
%***** we initialize our point-globals

global ptlen nrpgons expgonlen

ptlen = length(pt);
nrpgons = zeros(ptlen,1);
for i = 1:ptlen
    nrpgons(i) = size(pt{i}.pgon,2);
end

troubles = find(nrpgons<3);
if ~isempty(troubles)
    fprintf('
An error has occurred. The points: [%s]
    int2str(troubles));
    fprintf(' are contained in less than 3 polygons \n\n');
    error('Input Data Error');
end

function newpgon = fulledgeinfo(pt, pgon, orientation, or)
%***** To each polygon's edge, this function calculates the index of the
%***** adjacent polygon, and the index of the edge in this polygon.
%***** POLYGONS MUST BE ORIENTED NEGATIVELY

%***** data structure: newpgon{i} is a 3-by-nrpts+1 matrix, the first row
%***** containing the points of which the polygon consists, the second the
%***** Indices of the adjacent polygons, the third the indices of the
%***** edges
%***** in the adjacent polygons. Edges are 1-2, 2-3, ..., end-1.

global pgonlen ptlen nrpts nrpgons

for i = 1:pgonlen
    newpgon{i} = zeros(3,nrpts(i)+1);
    newpgon{i}(1,:) = pgon{i};
end
pgoni=pgon{i}; % speed
for j=1:nrpts{i}
  if newpgon{i}(2,j)==0 % this pgon has not already
    % been taken into account
      thispt=pgoni(j); % clearness
      nextpt=pgoni(j+1); % clearness
      adjgon = intersection(pt{thispt}.pgon(1,:),pt{nextpt}.pgon(1,:),i);
      % the index of the adjacent polygon
      % can be determined by intersecting the polygons
      % the first edge point with those containing the
      second.

      if isempty(adjgon)
        fprintf(‘
 An error has occurred at edge %i %n’,j);
        fprintf(‘ of polygon %i. I did not find an adjacent polygon.
’,j);
        error(‘Input Data Error’);
      end
      adjedge=findfirst(pgon{adjgon},nextpt);
      % Now get the index of the edge in the calculated
      % here we need the orientation of the polygons
      newpgon{i}(2:3,j)=[adjgon adjedge]; % and set the values
      newpgon{adjgon}(2,adjedge)=i;
      % we do not need the other indices of the adjacent
      pgon
      end % any more, so we leave them zero
end
end

function val=intersection(a,b,n)
%***** if a and b contain exactly two equal
%***** values, and one of them is given as n,
%***** this function calculates the other.
val=[];
for i=1:length(a)
  if a(i)==n
    f=b(find(b==a(i))); %
    if ~isempty(f)
      val=f(1);
      break;
    end
end
function first=findfirst(array, number)
%***** If array is known to contain number, this function returns
%***** the index of the first appearance of number in array.

i=1;
while 1
    if array(i) == number
        first=i;
        break;
    end
    i=i+1;
end

function out=costable(pgons)
%***** Calculates the weights for convex-combination of the polygon points

global nrpts nrpgons

for n=3:max([nrpts ; nrpgons ; 4]) % this number does not increase along iterations
    out{n}=zeros(n,n);
    w(1)=1/4*(1+5/n);
    for j=2:n
        w(j) = (3+2*cos(2*pi*(j-1)/n))/(4*n); % calculate the doo-weights for first point
    end
    out{n}(1,:) = w;
    for k=2:n % and now rotate them to get the doo-weights
        w=[w(n) w(1:n-1)]; % for the other points
        out{n}(k,:) = w;
    end
end

function [planeidx, vertidx, edgeidx, ptidx] = linearindexing(pt, pgon);
%***** Defines linear indices of both points and pgons of the next doo-step.

global pgonlen nrpgons nrpts

pgons4=(nrpgons==4); % we determine which points are ordinary
orptidx=find(pgons4); % and get their indices
orptlen=length(orptidx);

exptidx=find(~pgons4); % the same for extraordinary points
exptlen=length(exptidx);

expgonlen=length(find(nrpts==4)); % number of extraordinary polygons
orpgonlen=pgonlen-expgonlen;
lxlow=1; % linear index
lxhigh=expgonlen+1; % the first pgons derive from extraordinary
planepgons.
planeidx=lxlow:lxhigh-1; % they are extraordinary themselves

lxlow=lxhigh; % then we reserve space for the extraordinary
lxhigh=lxlow+exptlen; % pgons that derive from extraordinary vertices.
vertidx(exptidx)=lxlow:lxhigh-1;

lxlow=lxhigh; % all extraordinary pgons have been taken into
account,
lxhigh=lxlow+orpgonlen; % next polygons are those from ordinary pgons
planeidx=[planeidx lxlow:lxhigh-1];

lxlow=lxhigh; % polygons from ordinary vertices
lxhigh=lxlow+orptlen;
vertidx(orptidx)=lxlow:lxhigh-1;

lx=lxhigh; % polygons that derive from edges are last
edgeidx=cell(pgonlen,1);
for i=1:pgonlen
    edgeidx{i}=zeros(2,nrpts(i)+1);
end

for i=1:pgonlen
    pgoni=pgon{i}; % speed
    pgoni2=pgoni(2,:);
    pgoni3=pgoni(3,:);
    
    for j=1:nrpts(i)
        if edgeidx{i}(1,j)==0 % this pgon has not already been taken into
            edgeidx{i}(:,j)=[lx ; 4]; % this edgepgon derives from the
            % j-th edge of the pgon i
            % the edgepolygon starts at the
            % point, and therefore its fourth edge
            % will be adjacent to the shrunken
            pgon i
            
            edgeidx{pgoni2(j)}(:,pgoni3(j))=[lx 2];
            % the appropriate settings for the adjacent pgon
            lx=lx+1;
    end
end
end

ptlx=1; % while the linear indexing of pgons looks rather
for i=1:pgonlen % complicated, linear indexing of points is very
simple
function [ptnew,ptnewgon]=planepolygons(pt,pgon,planeidx,edgeidx)

***** Calculates the new points of a doo-sabin step, and the polygons, that
***** derive from the polygons of the previous step by shrinking.

global pgonlen ptlen nrpts sumnrpts nrpgons w

newpgon=cell(pgonlen+ptlen+sumnrpts/2,1); % #planepgons + #vertpgons +
#edgepgons

newpt=ones(sumnrpts,5); % note that we change the data-structure of the
points. % this is doo to the fact, that all new points are
% ordinary (connect 4 edges). The first three
entries % are the points coordinates, the fourth is a pgon
% that contains the point (no matter which one)
and % the fifth is the index of the point in that pgon

lx=1;
for i=1:pgonlen
nrptsi=nrpts(i);
   % speed
lxhigh=lx+nrptsi-1;
newpgon{planeidx(i)}={[lx:lxhigh lx] ; edgeidx{i}};
% the shrunken polygon is found easily
    p=zeros(nrptsi,3);
    % now for the new points:
    for j=1:nrptsi
        % we gather the pgon's point's coordinates in p
        p(j,:)=pt{pgon{i}(j)}/coos;
    end
    newpt(lx:lxhigh,:)=[w(nrptsi)*p planeidx(i)*ones(nrptsi,1)
[lx:lxhigh')]';
% and convex-combine them using the weights w
lx=lxhigh+1;
end

function
[newpt,newpgon]=vertexpolygons(pt,pgon,newpt,newpgon,vertidx,edgeidx,ptidx)
%***** Calculates the polygons that derive from cutting off the vertices.
%***** This algorithm may look a bit clumsy, but consider that we do not
%***** know the order in which to connect the points that form our new
%***** Vertex polygon. The points themselves are easily found.

global ptlen nrpts nrpgons

for i=1:ptlen
    nrpgonsi=nrpgons(i);
    vertpgoni=zeros(3,nrpgonsi+1);  % speed

    pgonnr =pt{i}.pgon(1,1);  % pgonnr stands for a the index of a
    pgonptnr=pt{i}.pgon(2,1);  % containing the point i. We start
    arbitrarily
    if pgonptnr==1
        edgenr=nrpts(pgonnr);  % pgonptnr is the index of the current
    point
    else
        edgenr=pgonptnr-1;  % in this pgon, and edgenr is the number
    of
    end

    for j=1:nrpgonsi-1
        vertpgoni(1:3, j)=ptidx(pgonnr)(pgonptnr);
        edgeidx(pgonnr)(1,edgenr);  % the edge leading to this point.
        edgeidx(pgonnr)(2,edgenr)-1;  % stores the information about the j-th vertex of the
        % vertex polygon in vertpgoni. First are the points the
        % vertex polygon consists of, second the adjacent pgons
        % and third the edge indices.

        pgonnr=pgon(pgonnr)(2,edgenr);
        pgonptnr=findfirst(pgon(pgonnr)(1,:),i);

        % The new pgonnr is the number of the polygon adjactent
        % to the old pgonnr along the edge edgenr. This polygon
        % must contain our point i, so we search for its index.
        % Doing this nrpgonsi times we encircle our vertex i and
        % find our vertex polygon.

        if pgonptnr==1
            edgenr=nrpts(pgonnr);
        else
            edgenr=pgonptnr-1;
        end
    end
end

vertpgoni(1:3,nrpgonsi)=...
    [ptidx(pgonnr)(pgonptnr); edgeidx(pgonnr)(1,edgenr);...
     edgeidx(pgonnr)(2,edgenr)-1];
newpgon {vertidx(i)} = vertpgoni;
newpgon {vertidx(l) + 1} = vertpgoni(l, 1);

function newpgon = edgepolygons (newpt, newpgon, pgon, edgeidx, vertidx, ptidx)
%***** Calculates the polygons that derive from cutting off the edges.

global pgonlen nrpts sumnrpts
edgepgon = cell (sumnrpts / 2, 1);

for i = 1 : pgonlen
  pgoni = pgon{i};
  pgoni1 = pgon{1,:};
  pgoni2 = pgon{2,:};
  pgoni3 = pgon{3,:};
  ptidxi = ptidx{i};

  for j = 1 : nrpts(i)
    pgoni3j = pgoni3(j);
    if pgoni3j == 0
      pgoni2j = pgoni2(j);

      ptidxpgoni2j = ptidx{pgoni2j};

      % share edge j with a pgon of lower lx
      % (in fulledgeinfo)
      newpgon{l} =
        [ptidxi(j) ptidxpgoni2j(pgoni3j+1)...
         ptidxpgoni2j(pgoni3j) ptidxi(j+1)]
        vertidx(pgoni1{j}) 0 ...
        vertidx(pgoni1{j+1}) 0 0];
    end
  end
end
end
end
end
end

function qlinearindexing;
%***** Defines linear indices of both points and pgons of the next doo-
step.
%***** Quick version, taking advantage of improved data structure

global pt ptlen nrpts pgon pgonlen nonedgepgonlen vertidx edgeidx ptidx
% we do not need planeidx anymore, since
planeidx(1) = i
vertidx = pgonlen + 1: lx; % so we start with vertidx

lx = lx + 1;

edgeidx = cell(pgonlen, 4);

for i = nonedgepgonlen + 1: pgonlen % then edgeidx
    edgeidx(i) = 2 * ones(2, nrpts(i) + 1); % the last coordinate is dummy (concat
end % the edge indices are known to
% be equal to 2 (sketch).

for i = 1: nonedgepgonlen
    nrpts = nrpts(i);
    pgon = pgon(i);

    edgeidx(i) = [lx:lx + nrpts, 4 * ones(1, nrpts + 1)];
    % this edgepgon derives from the
    % j-th edge of the pgon i
    % the edge polygon starts at the
    % point, and therefore its fourth edge
    % coincides with the current.

    for j = 1: nrpts
        edgeidx(pgon(2, j)) {1, pgon(3, j)} = lx;
        lx = lx + 1; % the appropriate settings for the adjacent pgon
    end
end

ptlx = 1;
for i = 1: pgonlen % linear indexing of points stays the
    pltxhigh = pltx + nrpts(i) - 1; % same.
    ptidx(i) = [pltx:pltxhigh, pltx];
    pltx = pltxhigh + 1;
end

function gplanepolygons
%***** Calculates the new points of a doo-sabin step, and the polygons,
% that
%***** derive from the polygons of the previous step by shrinking.
%***** Quick version, taking advantage of improved data structure

global pt ptlen pgon pgonlen nrpts newpt newpgon edgeidx sumnrpts w

newpgon = cell(pgonlen + ptlen + sumnrpts/2, 1); % # planepgons + # vertpgons +
# edgepgons
newpt = ones(sumnrpts, 5);
56

\[ \text{linear index} \]

for \( i = 1: \text{pgonlen} \)
\[
\text{nrptsi} = \text{nrpts}(i);
\]
\[
lxhigh = lx + \text{nrptsi} - 1;
\]
\[
\text{newpgon}(i) = \lfloor [lx:lxhigh lx] \rfloor; \quad \text{edgeidx}(i); \quad \% \text{planepolygon easily found}
\]
\[
p = \text{pt}(\text{pgon}(i)(1,1: \text{nrptsi}),1:3); \quad \% \text{We gather the pgon's point's coordinates in p}
\]
\[
\text{newpt}(lx:lxhigh,:) = \lfloor (w(\text{nrptsi})*p)*\text{ones}(\text{nrptsi},1) \rfloor; \quad \% \text{and convex-combine them to the new points}
\]
\[
lx = lxhigh + 1;
\]
end

function qvertexpolygons
\% **** Calculates the polygons that derive from cutting off the vertices.
\% **** This algorithm may look a bit clumsy, but consider that we do not
\% **** know the order in which to connect the points that form our new
\% **** vertex polygon. The points themselves are easily found.
\% **** Quick version, taking advantage of improved data structure

global pt ptlen pgon nrpts newpgon edgeidx vertidx ptidx

for \( i = 1: \text{ptlen} \)
\[
\text{vertpgoni} = \text{zeros}(3,5);
\]
\[
\text{pgonnr} = \text{pt}(i,4); \% \text{Due to the changed data format}
\]
\[
\text{pgonptnr} = \text{pt}(i,5);
\]
\[
\text{if} \quad \text{pgonptnr} == 1
\]
\[
\text{edgenr} = \text{nrpts}(\text{pgonnr});
\]
\[
\text{else}
\]
\[
\text{edgenr} = \text{pgonptnr} - 1;
\]
\[
\text{end}
\]

for \( j = 1:3 \)
\[
\text{vertpgoni}(1,j) = \text{ptidx}(\text{pgonnr})(\text{pgonptnr});
\]
\[
\text{vertpgoni}(2,j) = \text{edgeidx}(\text{pgonnr})(1,\text{edgenr});
\]
\[
\text{pgonnr} = \text{pgon}(\text{pgonnr})(2,\text{edgenr});
\]
\[
\text{pgonptnr} = \text{findfirst}(\text{pgon}(\text{pgonnr})(1,:),i);
\]
\[
\text{if} \quad \text{pgonptnr} == 1
\]
\[
\text{edgenr} = \text{nrpts}(\text{pgonnr});
\]
\[
\text{end}
\]
else
    edgenr=pgonptrn-1;
end
end

vertpgoni(1,4)=ptidx{pgonn}(pgonptrn);
vertpgoni(2,4)=edgeidx{pgonn}(1,edgenr);

vertpgoni(3,1:4)=[3 1 3 1]; % due to the ordering of the polygons,
% we know this in advance (sketch)

newpgon{vertidx(i)}=vertpgoni;
newpgon{vertidx(i)}(1,5)=vertpgoni(1,1);
end

function gedgepolygons
%***** Calculates the polygons that derive from cutting off the edges.
%***** Quick version, taking advantage of improved data structure

global pgon nonedgepgonlen nrpts newpgon vertidx edgeidx ptidx

lx=edgeidx{1}(1);
for i=1:nonedgepgonlen
    pgoni=pgon{i};
    pgoni1=pgoni(1,:);
    pgoni2=pgoni(2,:);
    pgoni3=pgoni(3,:);
    ptidxi=ptidx{i};

    for j=1:nrpts(i)
        pgoni2j=pgoni2(j);
        pgoni3j=pgoni3(j);
        ptidxpxgoni2j=ptidx{pgoni2j};

        newpxgon{lx}...
        =[ptidxi(j) ptidxpxgoni2j{pgoni3j+1} ptidxi{j+1} ptidxi{j};
        ptidxi{pgoni1(j)} ptidxi{j+1} ptidxi{j};
        ptidxi{pgoni1(j+1)} ptidxi{j+1} ptidxi{j};
        ptidxi{j};

        lx=lx+1;
    end
end
end

function opgon=pgonorient(argpgon)
% PGONORIENT orients a closed surface of polygons
%
% Let pgon be a cell array of polygons, each polygon
% being a pointlist (of indices to points), then
% opgon=PGONORIENT(pgon) orients the polygons defined by
% pgon with respect to each other.
% More accurately, it reverses the order of some polygons

% such that in the outgoing surface each edge is run once
% in each direction. SURFACEMUST BE CLOSED AND ORIENTABLE.
% The first polygon is assumed to have the right orientation.
%
% Example(Tetraeder):
%
>>pgon=[[1 2 3] [1 2 4] [2 3 4] [1 3 4]]; %
>>newpgon=pgonorient(pgon);
%
% See also D00 PGONDISP PGONTRACE CHSURF

global opgon opgonlen onrpts or nrorpgons orientation
%***** To keep the recursive function orient simple, globals are used

opgon=argpgon; % pass the argument to global
opgonlen=length(opgon);

maxpt=zeros(opgonlen,1); % speed
onrpts=zeros(opgonlen,1); % speed
for i=1:opgonlen
    maxpt(i)=max(opgon{i});
    onrpts(i)=length(opgon{i}); % number of points in opgon{i}
end
ptlen=max(maxpt); % number of points altogether

pt=fullpointinfo(ptlen,opgon,opgonlen,onrpts);
%***** To each point, this function calculates the polygons
%***** Containing it.

opgon=addfirstpoint(opgon);
%***** Adds the first point of the pgon to the
%***** end of the pgon's pointlist (convenient when referencing)

opgon=fulledgeinfo(pt,opgon,opgonlen,onrpts);
%***** To each polygon's edge, this function calculates the index
%***** of the adjacent polygon.

or=zeros(1,opgonlen); % or=1 for oriented pgons, zero for not yet oriented
or(1)=1; % the first pgon is assumed oriented correctly
nrorpgons=1; % hence the number of oriented pgons is one
orientation='pos'; % If not, orient will set it to 'none'
orient(1); % this recursive function does all the work

for i=1:opgonlen
    opgon{i}=opgon{i}(1,1:onrpts(i)); % brings pgon into input form
end

if orientation(1)=='p'
    fprintf('
 Polyhedral mesh was already oriented. 
');
SUBROUTINES
*************************************************************************
*                        SUBROUTINES                                      *
*************************************************************************

function pt=fullpointinfo (ptlen, opgon, opgonlen, onrpts)
***** To each point, this function calculates the polygons
***** containing it.
pt=cell (ptlen, 1);
for i=l:ptlen
    pt{i}=[];
    % reserve space
end
for i=l:opgonlen
    for j=1: onrpts{i}
        pgonij=opgon{i}(j);
        % speed
        pt{pgonij}=[pt{pgonij} i];
    end
    % concatenate pgon-data i to
    % existing pt-data
end

function newpgon=addfirstpoint(opgon);
***** Adds the first point of the pgon to the
***** end of the pgon's pointlist (convenient when referencing)
for i=l:length(opgon)
    pgoni=opgon{i};
    newpgon{i}=[pgoni pgoni{1}];
end

function newpgon=fulledgeinfo (pt, opgon, opgonlen, onrpts)
***** To each polygon's edge, this function calculates the index of the
***** adjacent polygon.
***** datastructure: newpgon{i} is a 2-by-onrpts+l matrix, the first row
%***** Containing the points of which the polygon consists, the second the
%***** Indices of the adjacent polygons.

newpgon=cell(opgonlen,1);

for i=1:opgonlen
    pgoni=opgon(i); % speed
    nrptsi=onrpts(i); % speed
    row2=zeros(1,nrptsi); % speed

    for j=1:nrptsi
        thispt=pgoni(j); % speed, clearness
        nextpt=pgoni(j+1); % speed, clearness
        adjpgon = intersection(pt{thispt},pt{nextpt},i);

        % the index of the adjacent polygon
        % can be determined by intersecting the polygons
        containing
        % the first edgepoint with those containing the
        % second.
        if isempty(adjpgon)
            fprintf('
 An error has occurred at edge %i 
',j);% speed, clearness
            fprintf(' of polygon %i. Did not find an adjacent polygon.
',i);
            error('Input Data Error');
        end
        row2(j)=adjpgon; % and set the value
    end
    newpgon{i}=[pgoni ; [row2 row2(1)]] ;% speed, clearness
end

function val=intersection(a,b,n)
%***** if a and b contain exactly two equal
%***** values, and one of them is given as n,
%***** this function calculates the other.
val=[];
for i=1:length(a)
    if a(i)==n
        f=b(find(b==a(i))); % speed, clearness
        if ~isempty(f)
            val=f(1);
            break;
        end
    end
end
function orient(pgonnr)
%***** Actual orientation. Algorithm is as follows: Starting with the
%***** first pgon we orient its neighboring pgons, then the neighboring
%***** pgons of these if not already oriented and so on.

global opgon opgonlen onrpts or nrorpgons orientation

notoriented=[]; % neighbouring pgons that are not yet oriented
pgonpgonnr=opgon{pgonnr}; % speed

for j=1:onrpts(pgonnr)
    adjpgon=pgonpgonnr(2,j); % adjacent pgon
    if or(adjpgon)==0 % if adjacent pgon is not oriented
        notoriented=[notoriented adjpgon]; % put it in the list
        pgonadjpgon=opgon(adjpgon); % speed
        thispt=pgonpgonnr(1,j); % speed,clearness
        nextpt=pgonpgonnr(1,j+1); % speed,clearness
        idxnextpt=findfirst(opgon{adjpgon}(1,:),nextpt);
        %***** the index of the next point in the adjacent polygon
        if pgonadjpgon(1,idxnextpt + 1) ~= thispt % wrong orientation
            pgonadjpgon=fliplr(pgonadjpgon); % reverse order
            pgonadjpgon(2,:)=[pgonadjpgon(2,2:onrpts(pgonnr)+1) 0]; % correct edge
            information
            opgon{adjpgon}=pgonadjpgon; % and store the data
            orientation='none';
        end

        or(adjpgon)=1; % now the adjacent pgon is oriented
        nrorpgons=nrorpgons+1; % and the number of oriented pgons
        % increases by one.
    end
end

if nrorpgons ~=opgonlen % if not already finished
    for j=notoriented % orient the neighboring pgons
        orient(j);
    end
end

function first=findfirst(array,number)
%***** If array is known to contain number, this function returns
%***** the index of the first appearance of number in array.

i=1;
while 1
    if array(i)==number
        first=i;
        break;
    end
    i=i+1;
end

function varargout=pgondisp(pt,pgon)
% PGONDISP Display polyhedral mesh.
% [LI, PAT]=PGONDISP (PT, PGON) displays the polyhedral mesh defined
% by PT and PGON and returns a light handle LI and an array of
% patch handles PAT.
% PT should be an m-by-3 matrix containing the coordinates of the
% points, PGON should be a cell array of polygons, each polygon
% being a point list(of indices to points).
% Example(Dice):
% pt=[0 0 0;1 0 0;1 1 0;0 1 0;...
%    0 0 1;1 0 1;1 1 1;0 1 1];
% pgon =[[1 2 3 4] [5 8 7 6] [1 5 6 2]...
%       [3 7 8 4] [2 6 7 3] [4 8 5 1]]; 
% pgondisp(pt,pgon);
% See also DOO PGONTRACE PGONORIENT CHSURF

%*****************************************************************************
% INPUT ARGUMENT CHECK %*****************************************************************************
error(nargchk(2,2,nargin));
if size(pt,2)==3
    pt=pt';
end
if size(pt,2)==3
    error(' pt should be an m-by-3 matrix');
end
%***************************************************************************** GRAPHICS %*****************************************************************************
axis off;
view(17,14);
rotate3d on;
pat=zeros(1,length(pgon));
for i=1:length(pgon)
ptpgoni=pt(pgon{i},:);  % speed
pat(i)=patch(ptpgoni(:,1), ptpgoni(:,2), ptpgoni(:,3),[1.0000
0.7812 0.4975],...
'facelighting','flat','edgelighting',[0 0 0],...
'edgelighting','flat','backfacelighting','lit');
end
li=light;
material dull;

%**************************************************************************
% OUTPUT ARGUMENT CHECK ***********************************************
%**************************************************************************
switch nargout
  case 1
    varargout{1}=li;
  case 2
    varargout{1}=li;
    varargout{2}=pat;
end

function varargout=pgontrace(pt,pgon,colorfile)
% PGNONTRACE Raytraces a polyhedral mesh
% [LI, PAT, IVN]=PGONTRACE(PT, PGN[, COLORFILE]) ray traces the
% Polyhedral mesh defined by PT and PGN using Vertex normal-
% Interpolation and returns an array of light handles LI, an
% Array of patch handles PAT and the interpolated
% vertex normal IVN.
% PT should be an m-by-3 matrix containing the coordinates
% of the points, PGN should be a cell array of polygons, each
% polygon being a point list (of indices to points).
% COLORFILE is the name of an m-file that defines the color of
% a point on the surface in dependency of its coordinates
% and its normal vector. See COLORFILE for details.
% See also COLORFILE DOO PGONDISP

%**************************************************************************
% INPUT ARGUMENT CHECK ***********************************************
%**************************************************************************
error(nargchk(2,3,nargin));
if size(pt,2)==3
  pt=pt';
end
if size(pt,2)==3
  error(' pt should be a m-by-3 matrix');
end
if nargin==2
colorfile='defaultcolormethod';
end

%******************************************************************************
% GRAPHICS
%******************************************************************************

map=get(gcf, 'colormap');   % set finer colormap
newmap=map;
if size(map,1)<=64
    for i=1:size(map,1)-1
        newmap(4*i-3:4*i,:)=[map(i,:); 0.75*map(i,:)+0.25*map(i+1,:);...
                        0.50*map(i,:)+0.50*map(i+1,:);...
                        0.25*map(i,:)+0.75*map(i+1,:)];
    end
end
set(gcf, 'colormap',newmap);

cla;
axis off;       % some graphic settings
set(gca, 'Projection', 'perspective');
view(17,14);
rotate3d on;
li=light;

pat=zeros(length(pgon),1);   % patch handles
vn =cell(1,length(pgon));    % vertex normals
for i=1:length(pgon)
    pgoni=pgon{i};       % speed
    ptpgoni=pt(pgoni,:);  % speed
    pat{i}=patch(ptpgoni(:,1), ptpgoni(:,2), ptpgoni(:,3),[0 0 0],...% sum(ptpgoni),'
    'FaceLighting','Phong','EdgeColor','None',...
    'EdgeLighting','None','Backfacelighting','unlit');
    vni=get(pat{i},'VertexNormal'); % get Vertexnormals
    for j=1:size(pgoni,2)
        vnij=vni(j,:);
        vn{i}(j,:)=vnij/norm(vnij); % normalize Vertexnormals
    end
end
ptdata=fullpointinfo(pt,pgon);

ivn=zeros(length(pt),3);     % interpolated Vertexnormal
colsize=length(feval(colorfile,[0 0 0],[1 1 1]));
col=zeros(length(pt),colsize);  % color of pt
for i=1:length(pt)
    ptdatai=ptdata{i};        % speed
    ivn=i
    col(i,:)=col(i,:)
end
ivni=vn{ptdatai(1,1)}{ptdatai(2,1),:}; % first vertex normal
for j=2:size(ptdatai,2)
    ivni=ivni+vn{ptdatai(1,j)}{ptdatai(2,j),:}; % add Vertex normal belonging
end
ivni=ivni/norm(ivni); % to the same point
ivn(i,:)=ivni; % normalize, and store them

col(i,:)=feval(colorfile,pt(i,:),ivni); % get color of pt
end

%************************* OUTPUT ARGUMENT CHECK ***************************
%***********************************************************************

switch nargout
    case 1
        varargout={li};
    case 2
        varargout={li pat};
    case 3
        varargout={li pat ivn};
end

%*******************************************************************************
%*******************************************************************************

function newpt=fullpointinfo (pt, pgon)
%***** To each point, this function calculates the polygons
%***** Containing it, and the indices of the point in these polygons.
newpt=cell(length(pt),1);

for i=1:length(pgon)
    pgoni=pgon{i}; % speed
    for j=1:size(pgoni,2) % speed
        pgonij=pgoni(j);
        newpt(pgonij)=[newpt(pgonij) [i;j]]; % concatenate pgon-data([i,j]) to
        end % existing pt-data
end

function color=defaultcolormethod (pt, nv)
% ***** If no colorfile is specified, defaultcolormethod is used.
color=sum(nv);