AUTOMATED METHOD FOR N-DIMENSIONAL SHAPE DETECTION BASED ON MEDIAL IMAGE FEATURES

by

Vikas Revanna Shivaprabhu

B.E., Visveswaraya Technological University, 2008

Submitted to the Graduate Faculty of

Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Vikas Revanna Shivaprabhu

It was defended on

December 02, 2010

and approved by

Co-advisor: Dr. George Stetten, Professor, Department of Bioengineering

Co-advisor: Dr. C.C.Li, Professor, Department of Electrical and Computer Engineering

Dr. Pat Loughlin, Professor, Department of Bioengineering

Dr. Marlin Mickle, Professor, Department of Electrical and Computer Engineering

AUTOMATED METHOD FOR N-DIMENSIONAL SHAPE DETECTION BASED ON MEDIAL IMAGE FEATURES

Vikas Revanna Shivaprabhu, M.S.

University of Pittsburgh, 2010

The focus of my thesis is to build upon the method of Shells and Spheres developed in our laboratory. The method as previously implemented extracts medial points based on the divergence of the direction function to the nearest boundary as it changes across medial ridges, and reports the angle between the directions from the medial point to two respective boundary points. The direction function is determined by analyzing the mean and variance of intensity within pairs of adjacent circular regions in a 2D image. My thesis research has involved improving the search method for determining the distance function and identifying medial points, and then clustering those medial points to extract features including scale, orientation and medial dimensionality. These are then analyzed to detect local geometric shapes. I have implemented the methods in *N* dimensions in the Insight Toolkit (ITK). In 3D, the method yields three fundamental dimensionalities of local shape: the sphere, the cylinder, and the slab, which, along with scale and orientation, are powerful features for classifying more complex shapes. Tests are performed on simple geometric objects including the hollow sphere (slab), torus (cylinder) and sphere. The results confirm the capability of the system to successfully identify the described medial shape features, and lay the foundation for ongoing research in identifying more complex anatomical objects in medical images.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I would like to sincerely thank my advisor Dr. George Stetten for all his support and guidance. This thesis would not have materialized without his continuous encouragement and motivation. George, thank you for having confidence in me and for showing me the intricacies involved in research work. It is an honor for me to be working with you.

I would like to thank the members of VIA lab for their input, especially Dr. John Galeotti for introducing me to ITK and helping me with the code. I appreciate Dr. C.C. Li for his guidance in getting my graduate career on the right track. I would also like to thank the members of my thesis committee for their precious time and suggestions.

I would like to thank my beloved brother, Vivek and his wife, Shruthi, and all my friends for being there for me when I needed them the most. Thank you all for making my stay in the U.S a pleasant and enjoyable experience.

Finally, and most importantly, I would like to express utmost gratitude to my parents for their undying support, encouragement, and for their blessings.

1.0 INTRODUCTION

Medical imaging has come a long way since the discovery of X-rays by Roentgen in 1895. Over the years, many new imaging modalities have evolved, providing doctors and radiologists insight into the human body. The images produced by different modalities provide vital information necessary for diagnosis and treatment. Doctors rely on these images to identify, measure and functionally assess various structures. Most of the techniques for analysis currently used in clinics require manual examination by a radiologist. This process is tedious and time consuming, especially with large 3D datasets such as produced by MRI, which are generally examined one slice at a time. Hence, computerized image analysis, as a means to assist doctors to extract information with little or no manual intervention, is of critical importance. Most of the automated analyses that exist are unreliable because of the irregularities that are inherently present in the images, such as noise, variation in anatomical shape, discontinuous object boundaries, and varying imaging characteristics. Hence, a more robust and rapid automated system is needed.

Typically, shape detection depends on the ability to identify boundaries, which are used to determine the parent shape by grouping neighboring boundary points. Such local measurement techniques are susceptible to image noise and may be unstable. A different approach would be to group all boundary points globally, using geometric relationships. But this approach is computationally expensive and not practical for large data sets. A compromise between these

two approaches can be achieved by considering the medial relationship, which links opposing boundary points equidistant from the center of an object. The framework of "shells and spheres", developed in Dr. Stetten's laboratory, is a means to perform this association by detecting medial points based on the statistical properties of populations of pixel intensities. This is accomplished by first determining the distance and direction to the nearest boundary at every pixel location. The work of this thesis is to build upon the framework of shells and spheres, examine its capability to detect and cluster medial points in *N*-dimensions, and extract features such as scale, orientation, and medial dimensionality. These features are analyzed to differentiate three fundamental local geometric shapes: sphere, cylinder, and slab. Tests have been performed on simple geometric objects including the hollow sphere (slab), torus (cylinder) and sphere. The methods have also been applied in a preliminary way to real 3D medical imaging data.

Thesis Overview:

i. A robust system capable of detecting medial manifolds in *N*-dimensions is developed.

ii. The local properties of the medial manifold, including scale, orientation, and dimensionality, are extracted for shape analysis.

iii. Evaluation of these methods on synthetic test objects (sphere, cylinder, and slab) confirms the ability of the system to detect medial manifolds and to identify their local scale, orientation, and dimensionality. Preliminary examination of the system on real 3D medical data is performed.

2.0 BACKGROUND

2.1 MEDIAL AXIS

The concept of the Medial Axis, originally referred to as topological skeleton, dates back to 1967 when Blum introduced it as a tool for biological shape recognition [1]. The medial axis is formally defined as the locus of centers of spheres contained within the object that are tangent to the boundary in at least two places. It was originally suggested as an effective means of representing objects in 2D images. A classic illustration of the Blum medial axis of a rectangle is shown in Figure 1. The dotted lines represent the locus of points equidistant from two or more boundary points of the rectangle. Also shown are a few of the circles whose centers lie on the medial axis and whose circumferences touch but do not cross the rectangle's boundary. Subsequently, Blum also suggested the extension of medial loci to objects in 3D images, using maximal spheres instead of circles. A useful metaphor for the Blum medial locus is the "grassfire", in which the medial locus is obtained as a set of quench points when a field of uniformly dense grass whose boundary matches the boundary of the object is set on fire at each point on the boundary at time $t = 0$.

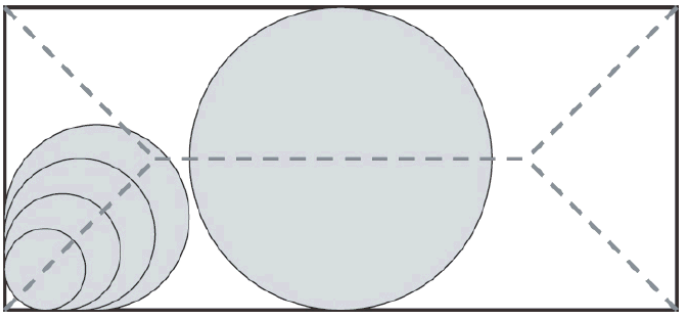

Figure 1: Blum medial axis (dotted lines) of a rectangle. It is locus of points equidistant from two or more boundary points of the rectangle (courtesy of Aaron Cois, Ph.D. Dissertation).

Blum's approach presupposes an existing segmentation, a binary image. Detection of the medial axis in gray scale images was implemented by Burbeck and Pizer [2]. They designed a "core" model in which a figure's boundaries are related to one another at a scale determined by the figure's width, as determined by statistical operations at that scale. A core is a locus in a space whose coordinates are position, radius, and associated orientations. The extraction of boundary and medial ridges using the core model has proved to be stable against image disturbances [3]. A framework that finds pairs of boundary points called "core atoms" using one such statistical approach has been implemented by Stetten and Pizer [4]. In core atoms, pre-detected boundary points are associated in pairs that face each other across an object and are then grouped by their centers into populations that are clustered at medial locations.

The medial representation has a variety of strengths. It is powerful since it directly captures various aspects of shape by giving direct access to both object interiors and object boundaries and also provides important features such as location, orientation and scale in any neighborhood of the interior of an object. The medial axis is a transformation of an object boundary with the same topology as the object. It is not only possible to generate the medial axis

from the boundary but equally, the medial locus can generate the object boundary. As a result of such advantages, medial representation finds wide use in image analysis, computer vision and other fields of computer science.

2.2 SHELLS AND SPHERES

Shells and Spheres is a novel system developed in our laboratory for analyzing images. It is based on a set of spheres, one centered at each pixel in an image, whose radii are allowed to grow. A *sphere map* is an *N*-dimensional neighborhood of pixels that lie within a radius *r* of a center point. Thus, $S_r(x) = \{y : \text{round}(|y - x|) \leq r, y \in \Omega\}$, where, $S_r(x)$ is a sphere of radius *r* centered at image pixel *x*, *y* is a pixel within that sphere, and $\Omega \subset Z_N$, is the set of all pixel locations in a sampled *N*-dimensional image. A s*hell* is the set of all pixels whose distance to the center rounds to a given radius, defined for a radius *r* as $H_r(x) = \{y : \text{round}(|y - x|) = r, y \in \Omega\}$. Figure 2 depicts a *sphere map* of radius 2 (shown in blue) and a *shell* of radius 3 (shown in red). Each pixel is shown as a number indicating its integer distance from the central pixel.

6	5	4	4	4	4	4	5	6
5	4	4	3	3	3	4	4	5
4	4	3	2	2	2	3	4	4
4	3	2	1	1	1	2	3	4
4	3	2	1	0	1	2	3	4
4	3	2	1	1	1	2	3	4
4	4	3	2	2	2	3	4	4
5	4	4	3	3	3	4	4	5
6	5	4	4	4	4	4	5	6

Figure 2: Each pixel is shown as a number indicating its integer distance from the central pixel. If we denote the central pixel as *x*, then pixels labeled *n* are members of the set $H_n(x)$. For example, the pixels labeled "3" (shown in red) comprise the shell $H_3(x)$.

A sphere of radius r can be formed from a union of shells,

$$S_r(x) = \bigcup_{k=0}^{r} H_k(x)$$

A sphere is allowed to grow such that it touches but does not cross the boundary of the object. Figure 3 shows a noiseless image with 2 objects having pixel intensities of '1' and '9' respectively. The boundary between the objects is shown by a dotted line. At pixel x, sphere $S_r(x)$ is allowed to grow till it reaches the boundary. Similarly, sphere $S_r(y)$ centered at pixel y is allowed to grow in the neighboring object. Since $S_r(x)$ and $S_r(y)$ touch but do not cross the boundary, it can be said that the spheres are optimized correctly. The optimal spheres' radii are equivalent to a distance map, indicating the distance from the center of each sphere to the nearest boundary.

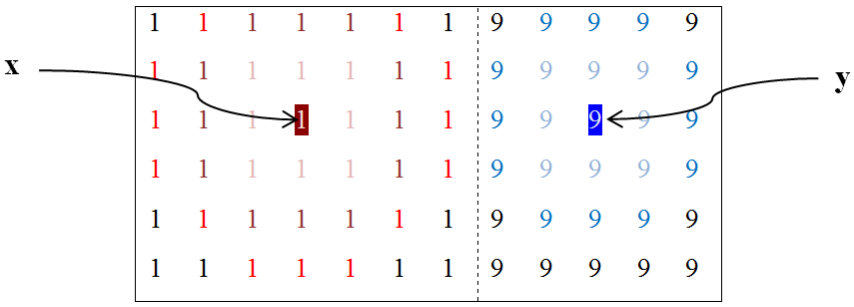

Figure 3: Noiseless image with boundary between two objects. Numbers indicate pixel intensity. The spheres are optimized to the right size. Spheres touch, but do not cross the boundary between the two objects. $S_r(x)$ has $r(x) = 3$ and $S_r(y)$ has $r(y) = 2$. Color intensity indicates the sphere growing from size 0, red for $S_r(x)$ and blue for $S_r(y)$.

This approach was taken prior to the present dissertation by Aaron Cois, in the Stetten laboratory, using Shells and Spheres to grow spheres from neighboring objects to meet at the

boundaries between them [5]. We now take a somewhat different approach, using asymmetric sphere pairs (described below), in which only one of the spheres is permitted to grow.

3.0 DETERMINING DISTANCE FUNCTION

The focus of my thesis is to build upon the framework of Shells and Spheres. Building on the previous work in our lab [5], I will utilize pairs of spheres that are adjacent, to tell the difference between the intensities of adjoining regions. *Sphere pairs* that are asymmetric, in terms of the radii of the individual spheres, can be used to determine the distance function, i.e., the distance to the nearest boundary, as well as the direction to that boundary. A *sphere pair*, as we define it, consists of a constant radius *outer sphere*, adjacent to an *inner sphere* whose radius can vary. The labels *inner* and *outer* denote the ideal placement of the sphere pair relative to an underlying object. The constant radius of the outer sphere is chosen to be small enough to provide sufficient boundary curvature, while still being large enough to represent a statistically significant population. The radius of the inner sphere is permitted to grow, starting at the same radius as the outer sphere, in search of the nearest boundary. The radius of the inner sphere is considered the *scale* of the sphere pair, and the sphere pair is said to be *located* at the center of the inner sphere.

We define various statistics for spheres. The *mean* intensity of the pixels within sphere $S_r(x)$ is defined by

$$\mu(x) = \frac{1}{|S_r(x)|} \sum_{y \in S_r(x)} f(y)$$

where $|S_r(x)|$ is the number of pixels in sphere $S_r(x)$ and $f(y)$ is the intensity at pixel y.

The *variance* at pixel x is defined as

$$\sigma^2(x) = \frac{1}{|S_r(x)| - 1} \sum_{y \in S_r(x)} [f(y) - \mu(x)]^2, \quad |S_r(\mathrm{x})| > 1$$

The *standard deviation* $\sigma(x)$ is the square root of the *variance*.

These statistics are first pre-computed for outer spheres at their constant radius for every pixel throughout the image. The statistics are then computed for the inner sphere at each pixel location, with the radius first set to that of the outer spheres and then allowed to increase until it reaches a maximum value set by the user, with the statistics recomputed for every radius. These statistics will be compared to those of the adjacent outer spheres to determine the presence of a boundary, as will be described below.

The radius of the outer sphere, which is also the initial radius of the inner sphere, must be at least 1. A sphere of radius 0 consists of a single pixel. If variance is to be computed, it is imperative that the radius of the sphere be greater than 0, since the variance of a sample of one pixel is not defined.

Boundaries between objects may be detected using the *d' (d-prime)* value, which is a statistic used in signal detection. It provides the separation between the means of the signal and the noise distributions, in units of the standard deviation of the noise distribution. In our case we have two populations of pixel intensities (those within the inner and adjacent outer spheres), and we want a measure of how different the two populations are, so as to be able to detect a boundary between the spheres. Hence, the signal and the noise distributions in the definition of *d'* can be replaced by the intensity distributions of the inner and the outer spheres. Since we care about the standard deviations of both populations, the formula for *d'* is given by:

$$d' = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}}}$$

where μ and σ are the means and standard deviations of populations 1 and 2.

Figure 4 demonstrates the concept of detecting boundaries using *d'*. The figure consists of a noisy image with a rectangular object whose mean pixel intensity differs from that of the background, with an inner sphere (red), and three adjacent outer spheres (blue) superimposed. Intensity histograms of the three sphere pairs (formed from the inner sphere and each of the three outer spheres) are shown to the right. The dotted line represents the histogram of the inner sphere, while the solid line represents the histogram of the outer sphere. The inner sphere encompasses an object completely within the object. Each of the sphere pairs is examined.

Sphere pair 1: The outer sphere encompasses a region completely outside the object. As seen in the corresponding histogram of pixel intensities (labeled 1), the means are clearly separated. The variance in each of the spheres is relatively low. Hence, a high *d'* results.

Sphere pair 2: The outer sphere includes pixels from both inside and outside the object. The means of the inner and outer spheres are closer to each other and the histograms (labeled 2) overlap. Also, the variance in the outer sphere is higher. Hence, a lower *d'* results.

Sphere pair 3: Both the inner and outer spheres are entirely within the object. The corresponding histograms (labeled 3) completely overlap. The means of the population in both the spheres are approximately the same. Hence, the *d'* is very close to 0.

A high *d'* indicates that the two spheres in a sphere pair lie on opposite sides of a boundary, with the point of contact between the two spheres lying on the boundary. Noticeably, in the example illustrated in Figure 4, sphere pair 1 generates the highest d' and is therefore the best choice for a boundary between the object and the background.

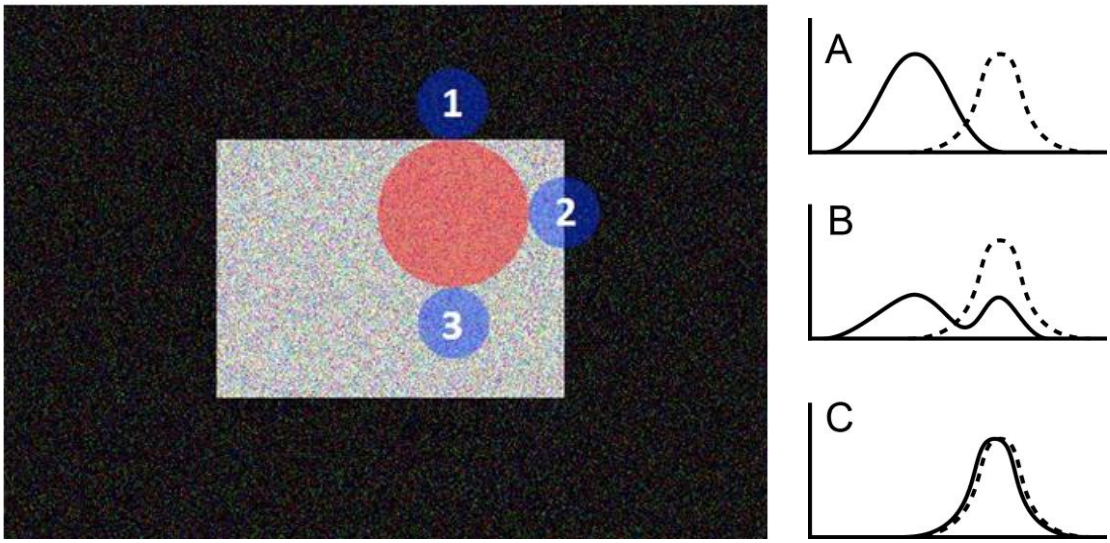

Figure 4: Noisy image consisting of 2 objects. A sphere pair consists of an inner sphere (red) and an outer sphere (blue). Three cases depicting different orientations of the outer sphere are shown, with their corresponding pixel intensity histograms (outer sphere = solid line, inner sphere = dashed line).

At a given pixel location, the inner sphere is allowed to grow, starting from a small radius equal to the radius of the outer sphere. As the inner sphere grows, sphere pairs are formed at every scale with all possible orientations of the outer sphere. For example, let us consider an inner sphere whose initial radius is 1 and let us assume that this sphere has now grown to a radius of 3, as illustrated in Figure 5. Sphere pairs (inner sphere + outer sphere) are now formed

that comprise the current inner sphere of radius 3 (red) and outer spheres that are located around, and adjacent to, the inner sphere (blue).

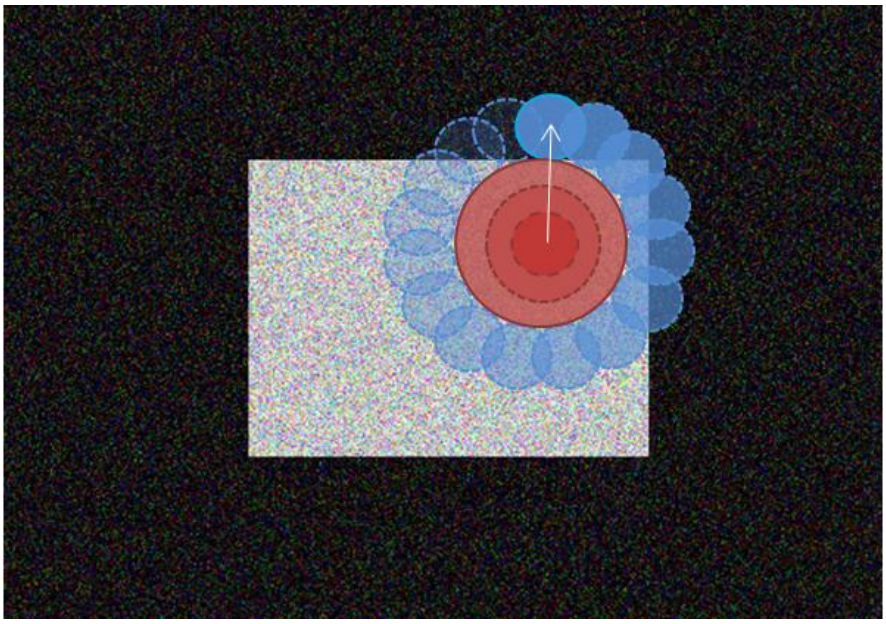

Figure 5: Inner sphere (red) is grown from radius 1 to radius 3. Sphere pairs are formed at all possible outer sphere (blue) orientations. The *d'* is computed for each sphere pair and the optimum sphere pair is chosen. The winning sphere pair provides the direction to the nearest boundary.

The *d'* value is computed for each sphere pair, and the one with the highest *d*' provides the optimum sphere pair at the current scale. For the example in Figure 5, the optimum sphere pair is most likely to include the outer sphere highlighted by a solid line. Assuming a correct optimization of the sphere pair by the method just described, we can obtain a unit direction vector $\hat{\mathrm{d}}(x)$ from the center of the inner sphere to the center of the outer sphere. (The ^ symbol denotes unit vector.) The direction vector, along with the corresponding *d'* and the current radius is stored using a method that is described in the following section. The inner sphere radius is then increased by 1 and the search for the optimum sphere pair is repeated at the new scale.

This process is repeated until the inner sphere reaches a maximum size set by the user. A direction vector is thus found at every scale, for each pixel location.

3.1 STORING THE DIRECTION VECTORS

As described in the previous section, a direction vector is obtained at each scale, and at each pixel location. Thus the direction vector is represented in a domain commonly known as "scale space". If the radius of the inner sphere ranges from 1 to 10, we obtain 10 different direction vectors pointing to the most likely boundary at each scale. Thus, we require a data structure capable of storing multiple records at each pixel location.

The data structure used in my thesis is a *linked list*. A linked list is a data structure that consists of a sequence of data records allowing easy insertion and removal of data. Each record of the linked list holds the scale, *d'*, and the direction, as shown in Figure 6.

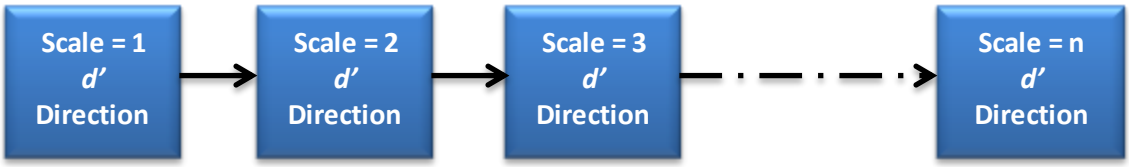

Figure 6: Linked list in which each record stores the scale, *d'*, and the direction

If a record were stored for every scale at every pixel, this would represent an enormous amount of data, for a typical 3D medical image. Luckily, as will be discussed in the following sections, only certain locations and scales are considered significant, namely those that are medial.

4.0 IDENTIFYING MEDIAL POINTS

As one crosses the medial ridge, the direction vector, which points to the nearest boundary, changes abruptly, as shown in Figure 7. By detecting such abrupt changes, we can identify medial points. Since the direction vectors are unit vectors, the dot product of two direction vectors gives us the cosine of the angle between them. This value ranges from -1 to +1. While positive values of the dot product imply that the two vectors are roughly in the same direction, negative values imply that they are in opposite directions. By identifying a pair of neighboring pixels whose direction vectors produce a negative dot product, it is possible to detect the abrupt change across the medial ridge, implying the existence of a medial point between the pixels. Note that in cases where the vectors form a right angle (see Figure 7) the dot product will yield 0. A threshold can be set to include any desired range of angles.

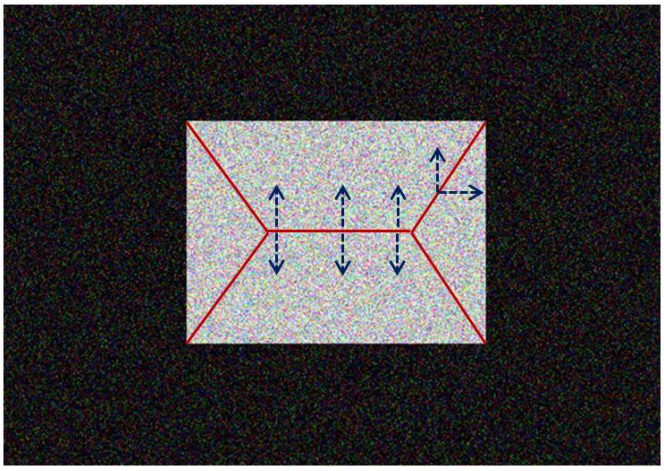

Figure 7: A noisy image showing the medial locus (red) of the object. Also shown is the direction of the sphere pair at a pixel location (blue).

The algorithm used to compute the dot product of the direction vectors of neighboring pixels is explained in the next section.

4.1 ALGORITHM TO DETECT MEDIAL POINTS

In terms of the data structure described in section 3.1, storing a record at every scale for every location would be redundant. It is sufficient to store a record only at the medial points and then only at the appropriate scale for that medial point.

In order to detect medial points in an image, a mask that includes the immediate neighbors of a given pixel in the positive direction in each dimension (x and y in 2D; x, y and z in 3D) is used. Figure 8 shows the 2D and 3D masks.

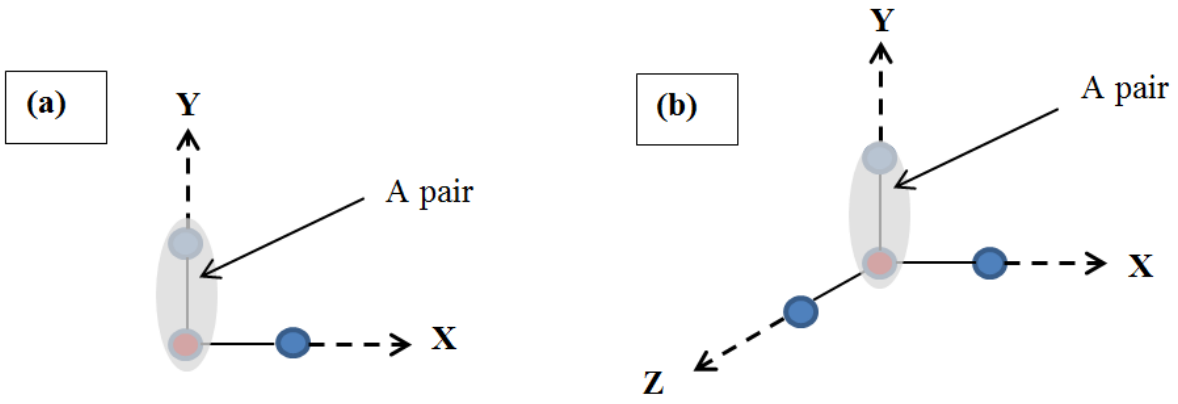

Figure 8: (a) 2D mask and **(b)** 3D mask used to detect medial points. A center pixel (red) and its immediate neighbors (blue) in the positive direction in each dimension are shown.

The mask is applied at every pixel location. Pairs of pixels are formed by considering the center pixel (red) and one of its neighbors (as defined by the mask). Thus, we obtain 2 pairs of pixels in 2D and 3 pairs in 3D at each pixel location. The dot product of the direction vectors is

computed for each pair. If the dot product is negative and satisfies a threshold set by the user, the pixels formulating the pair are marked as medial points, assuming that the *d'* associated with each direction vector is above a threshold, i.e., the corresponding sphere pairs have each located significant boundaries.

We apply this technique to detect medial points at every scale sequentially. Sphere pairs are first computed and stored temporarily at every pixel location of the image at a given scale. Medial points are then detected at this scale using the dot product of the direction vectors as just described. A record containing scale, the corresponding *d*', and the direction is stored in the linked list *only at the medial points*, instead of every pixel location. This process is repeated at each scale as the inner sphere is grown until it reaches its maximum radius.

To validate this method, tests were performed on three synthetic 3D objects: a torus, a hollow sphere, and a sphere. We chose these three objects because our aim is to detect the fundamental 3D shapes, namely, the cylinder, slab, and sphere. A torus is basically a cylinder locally with varied orientation, and a hollow sphere is a slab locally with varied orientation.

The results obtained are shown in Figures 9, 10, and 11. The images contain the synthetic 3D objects, which were generated using ITK (introduced in a later section). The dimensions of the images are 75x75x75 pixels and the voxel spacing is isometric. Gaussian noise has been added to the images. For each shape, medial points, which are expected to be related to the diameter of the shape at that point, are overlaid with the radius coded by color. A 3D surface rendering of each shape visualized using ITK-SNAP is also shown.

Figure 9 shows slices of a noisy 3D image containing a torus, whose minor radius is 6 and major radius is 15. The medial points detected at scale 1, 5, and 6 are shown in red, blue and green respectively. The majority of medial points are detected at scale 5. Fewer medial points are

detected at scale 6 (green), and they are therefore located far apart from each other. The points detected at scale 1 (red) are near the edge of the object. These points have been falsely identified as medial points due to an unresolved problem in the system.

Figure 10 shows slices of a noisy 3D image containing a hollow sphere of thickness equal to 3. Medial points are detected at scale 1 (red).

Figure 11shows the result obtained for a sphere of radius 8. As expected, the medial occupy a small region at the center of the sphere at scale 5 (blue).

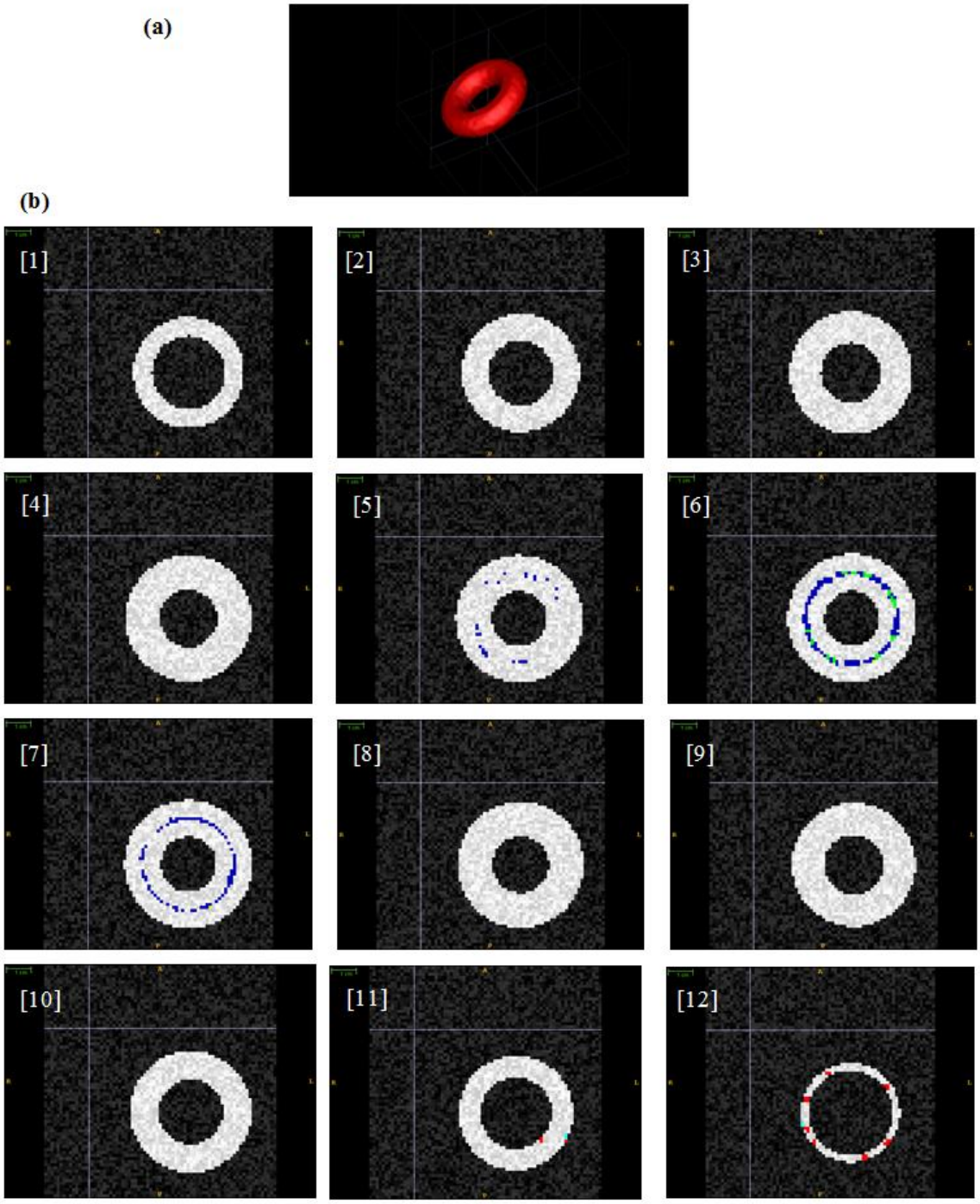

Figure 9: (a) 3D model of a torus visualized using surface rendering in ITK-SNAP. (b) Noisy image of a torus. Slices of the 3D image, showing the cross section of the torus and the detected medial points (slices 5, 6, 7, and 12) at scale 1 (red), scale 5 (blue), and at scale 6 (green)

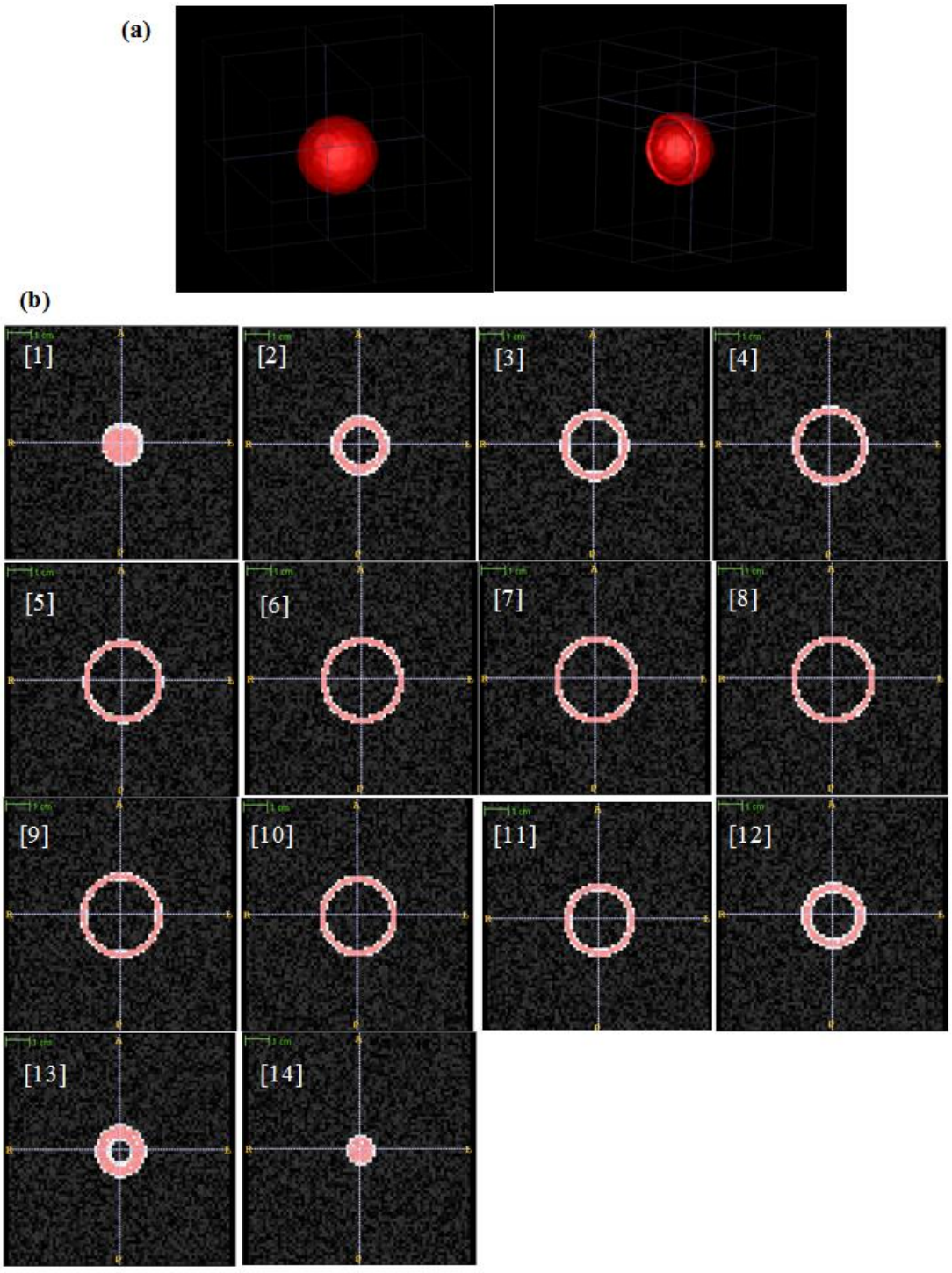

Figure 10: (a) 3D model and a cross sectional view of a hollow sphere. (b) Slices of a noisy 3D image of a hollow sphere, showing the cross section of the hollow sphere and the detected medial points at scale 1 (red)

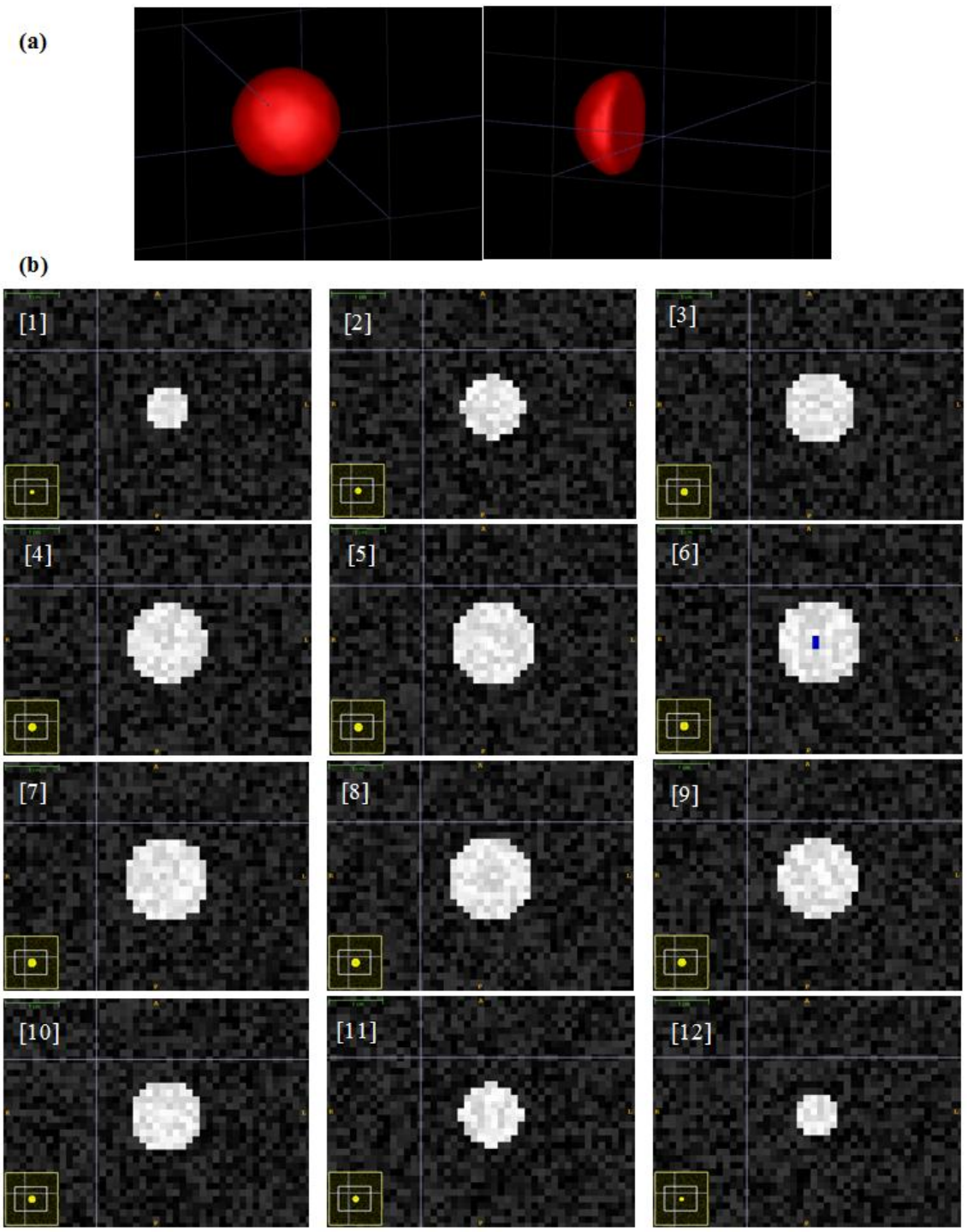

Figure 11: (a) 3D model and a cross sectional view of a sphere. (b) Slices of a noisy 3D image of a hollow sphere, showing the cross section of the hollow sphere and the detected medial points at scale 5(blue) in slice 6.

5.0 CLUSTERING MEDIAL POINTS

The medial points were clustered and analyzed to detect geometric shapes. Clustering was performed by describing a local region and a particular scale. The region was simply the inner sphere of any sphere pair at a medial location, as determined in the previous section. Any other medial points (sphere pairs) of the same scale within that inner sphere were considered to be in the cluster. Thus, a cluster $C_r(x)$ at pixel location x is defined as $C_r(x) = \{S_r(x) \cap M\}$, where $S_r(x)$ is an inner sphere of radius r centered at image pixel x, and M is the set of all medial points. For example, Figure 12 shows a noisy image containing an object. A set of medial points is shown in red. Let us say that a medial point at location x has been detected at scale 5 using the method described in the previous section. This means that at pixel location x an optimum sphere pair exists whose inner sphere is of radius 5, and a second sphere pair (not shown) is to the left of it pointing towards the opposite boundary. We can now form a cluster of medial points (dotted red line) that are within the inner sphere (blue) of radius 5 centered at x.

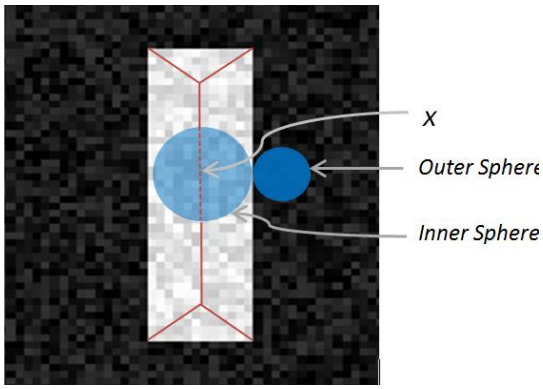

Figure 12: Medial points (dotted red line) within the inner sphere (blue) centered at x are clustered

6.0 EXTRACTING FEATURES

Local clusters of direction vectors are configured in three basic ways, corresponding to the fundamental geometric shapes of sphere, cylinder, and slab [5]. Each shape corresponds to a particular medial dimensionality, and has a particular distribution of direction vectors.

Sphere: The direction vectors point outward from the center towards the boundary in all possible directions. This can be compared to light rays originating from a point source of light. The center of the sphere is the medial point.

Cylinder: Medial points form the axis of the cylinder. The direction vectors point away from the axis to the nearest boundary within the plane orthogonal to the axis. This can be compared to the spokes of a wheel.

Slab: The medial points form a plane, with the direction vectors pointing away from the plane in opposite directions towards the nearest boundary.

These three configurations are illustrated in Figure 13. Note that the dimensionality of the particular medial manifold corresponds to the shape. Thus, for the sphere, cylinder, and slab, the medial manifold is the point, line, and plane, respectively. This corresponds to linear spaces with dimensionality 0, 1, and 2.

A given population of direction vectors may be examined to determine which of the basic three shapes it belongs to. Eigenanalysis is performed on the cluster of direction vectors by computing the covariance matrix D, which is given by

$$D = \frac{1}{n}\sum_{i=1}^{n} \hat{d_i}\hat{d_i}^T$$

where d_i is a population of n unit directional vectors.

Assuming we are working in N dimensions, we can obtain N eigenvalues from the covariance matrix D. The eigenvalues are denoted as $\lambda_1, \lambda_2 \ldots\ldots \lambda_N$ and their corresponding eigenvectors are denoted as $\hat{a}_1, \hat{a}_2 \ldots . \hat{a}_N$. Since D is a positive definite symmetric matrix, its eigenvalues are all positive and sum to 1, i.e. $\lambda_1 + \lambda_2 + \cdots + \lambda_N = 1$. If we arrange the eigenvalues such that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_N$, then their relative values can be used to represent dimensionality of the medial manifold, and their corresponding eigenvectors $\hat{a}_1, \hat{a}_2 \ldots . \hat{a}_N$ can represent its orientation. Eigenvector $\hat{a}_1$ will be the vector whose direction is most orthogonal to the population of direction vectors. Eigenvector $\hat{a}_N$ will be most collinear to the population.

In the case of 3D, λ_1 and λ_2 can be used to detect the dimensionality of an object. Note that it is sufficient to compute λ_1 and λ_2 since $\lambda_3 = 1 - \lambda_1 + \lambda_2$. As shown in Figure 13, for a cylinder, $\hat{a}_1$, being the most orthogonal to the population of direction vectors, represents the axis of the cylinder. The corresponding eigenvalues are $\lambda_1 = 0$ and $\lambda_2 = 1/2$. An eigenvalue of 0 implies that the corresponding eigenvector is completely orthogonal to every direction vector in the population, which is true for the spoke of a wheel configuration of direction vectors in the perfect cylinder. For a slab, both λ_1 and λ_2 are equal to 0. This implies that $\hat{a}_1$ and $\hat{a}_2$ are completely orthogonal to the population, i.e., the medial manifold is a plane running down the middle of the slab. In the case of a sphere, none of the eigenvectors are orthogonal to the population, which points out from the center in all direction equally. The analysis is summarized in table 1.

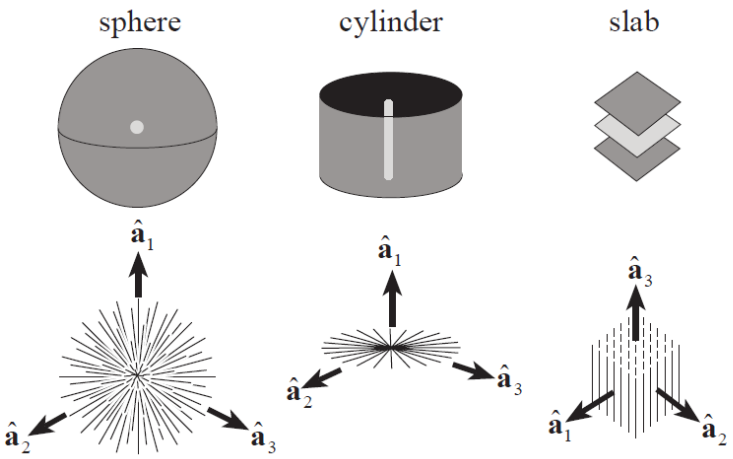

Figure 13: Three fundamental shapes and their corresponding medial manifolds. Also shown is the distribution of direction vectors in each shape and their corresponding eigenvectors. (Courtesy: George Stetten)

Table 1: Relation of eigenvalues to 3D shapes.

	Sphere	**Cylinder**	**Slab**
$\boldsymbol{\lambda_1}$	1/3	0	0
$\boldsymbol{\lambda_2}$	1/3	1/2	0
$\boldsymbol{\lambda_3}$	1/3	1/2	1

The relationship between the eigenvalues (λ_1 and λ_2) and the geometric shapes (sphere, cylinder, and slab) may be represented by a triangular domain we call the "lambda triangle" (Figure 11) [6]. As seen in Figure 11, the eigenvalues λ_1 and λ_2 defines the y-axis and x-axis respectively. All possible values of λ_1 and λ_2 fall within the triangle. The vertices of the triangle correspond to the 3 basic shapes (sphere, cylinder, and slab). Dimensionality may be approximated by arbitrarily dividing the triangle into three compartments, each representing a fundamental 3D shape. Given a population of direction vectors, the eigenvalues (λ_1 and λ_2) for

a given cluster can be plotted on the lambda triangle to identify the dimensionality depending on where the eigenvalues lie.

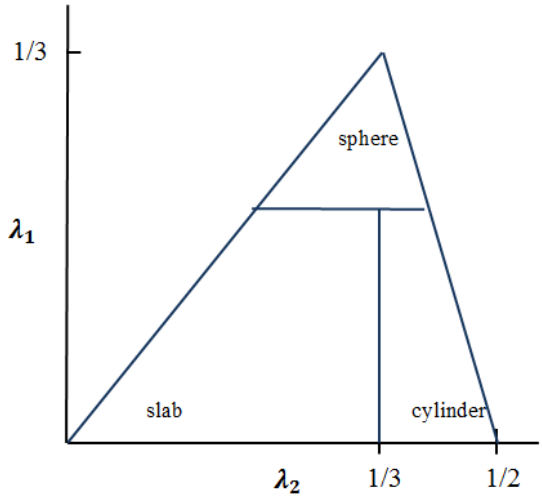

Figure 14: Triangle representing the relation between eigenvalues and 3D shapes

Figures 15 and 16 show scatter plots of the eigenvalues for the hollow sphere and the torus (described in section 4.1), computed using the methods described above. Color denotes scale. As shown in Figure 15 for the hollow sphere (Figure 10), medial points are detected only at scale 1. This is expected because the wall thickness of the hollow sphere is 3. (Recall that a sphere of radius 0 is a single pixel, and a sphere of radius 1 has a diameter of 3.) Figure 15 demonstrates that clusters formed at medial points of scale 1 contain sufficient number of direction vectors to perform meaningful eigenanalysis. Looking at the plot of the eigenvalues on the lambda triangle, we can see that the eigenvalues fall in the "slab compartment", thus confirming the dimensionality of the hollow sphere.

In the case of the torus (Figure 9), the scatter plot of the eigenvalues on the lambda triangle in Figure 16 shows medial points detected at scales 5 and 6, with the majority being at scale 6. Since the medial points detected at scale 5 (green, Figure 9) were fewer and located far apart from each other, the individual clusters formed at these locations do not contain sufficient numbers of direction vectors to produce meaningful eigenvalues, and thus the eigenvalues are distributed over the entire plot and not concentrated at any one region. On the contrary, at scale 6, the medial points that are detected are sufficiently numerous and close to each other to produce satisfactory eigenvalues, which are grouped near the vertex of the lambda triangle that corresponds to the cylinder.

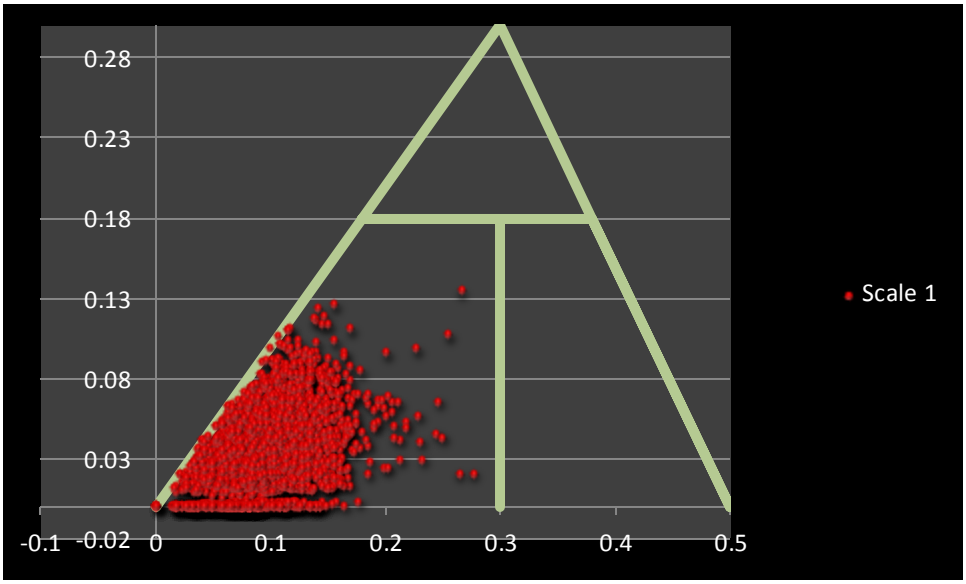

Figure 15: Eigenvalues of the hollow sphere superimposed on the lambda triangle

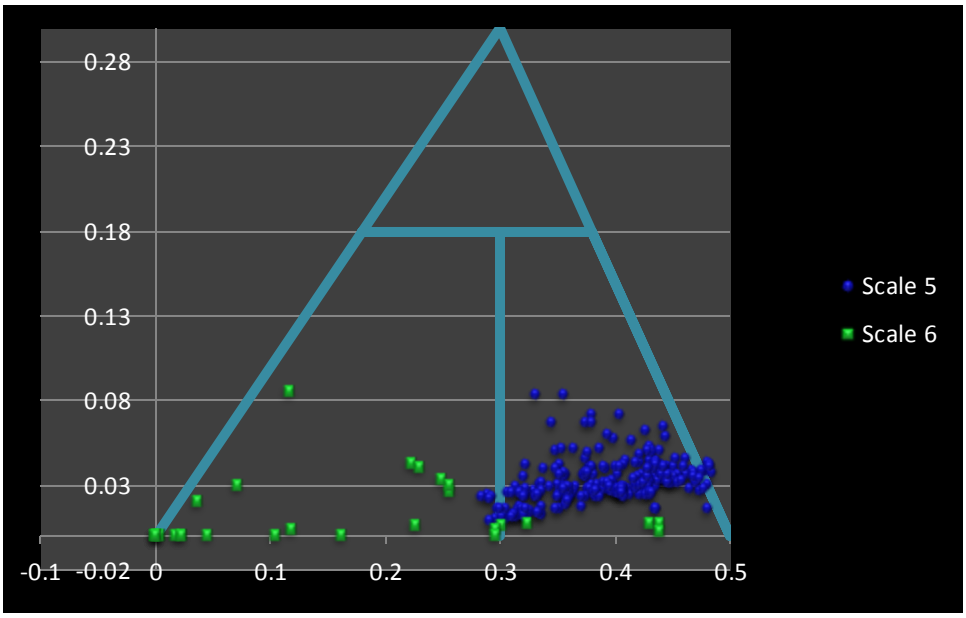

Figure 16: Eigenvalues of the torus superimposed on the lambda triangle

In the case of the sphere (Figure 11), each pixel location within the sphere has a direction vector that points outward towards the boundary of the sphere. The abrupt change of direction required to detect a medial point, as described in section 4.0, can be found only at the center of the sphere. Hence, medial points are detected at the center of the sphere, as shown in Figure 11. If we use the clustering method described in the previous section, the eigenvalues resulting from the cluster will not be meaningful, because only few medial points are available for clustering. We may address this in future work as follows: By altering the criteria used to cluster the direction vectors, more vectors could be included. Instead of clustering only the direction vector from the sphere pair with the greatest *d'* for each medial point, we could include in the cluster, direction vectors from sphere-pairs with somewhat lower *d'* (but still significant). There will be large numbers of these, especially at medial points in the center of spherical objects. This should yield sufficient number of samples to permit computation of eigenvalues, and we expect these values to be λ_1, λ_2, and λ_3 approximately equal to 1/3 for the sphere.

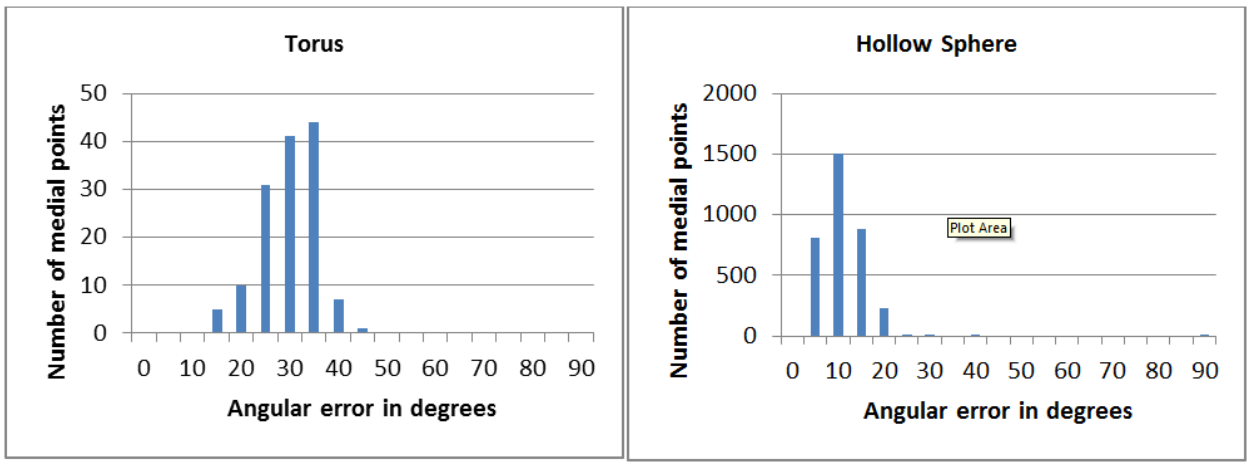

Figure 17: Orientation error in degrees for the torus and the hollow sphere.

We found the orientation error for the torus and hollow sphere, as shown in Figure 17. Error was computed by finding the theoretical axis of the local cylinder in the torus and the normal to the local slab of the medial manifold for the hollow sphere. The corresponding eigenvector for each medial pixel ($\hat{a}_1$ for the cylinder and $\hat{a}_3$ for the slab) was then compared to these theoretical values and the angular error reported. As can be seen in Figure 17, the error showed that the eigenvectors were consistently aligned in the correct orientation, within approximately 40 degrees for the torus and 20 degrees for the hollow sphere.

7.0 MEDIAL DETECTION IN A REAL IMAGE

The improved shells and spheres framework described in this thesis was tested on a real image. A 3D grey-scale lung data set of contrast-enhanced CT data, cropped and resampled to 75x63x99 isometric voxels, was used as the test image (Figure 18). The goal was to find medial points of the arterial and venous vessels, which constitute major structures in the lungs. The result obtained after applying the methods and algorithms described in the previous sections is shown in Figure 19. The radius of the outer sphere is equal to 1, and the maximum radius of the inner sphere is equal to 7. Although a maximum inner sphere radius of 3 would have been sufficient to detect the distal regions of the vasculature, the inner sphere was intentionally allowed to grow to a radius of 7, so that the robustness of the system to detect medial points at various scales could be tested. Our system identified medial points at scales ranging from 1 to 5, with the majority being detected at scale 1. Figure 19 shows a small selection of coronal, sagittal and axial slices of the 3D lung image, with the medial points superimposed on the image. The medial points have been color-coded to represent scale (see figure). As expected, the majority of medial points in the vasculature were detected at scale 1 (red). At places where the vessels are thicker, medial points were detected at higher scales. A flexibility in the system comes from the fact that the user can choose the scale at which he wishes to find the object. In our case, having prior knowledge that the vessels are small regions, we can opt to view the medial points at scales 1, 2, and 3, resulting in the removal of the unwanted medial points that would be detected at higher scales. The medial

points can then be rendered using various methods to obtain a 3D visualization. One such method using the open-source visualization software ITK-SNAP is shown in Figure 20. The figure shows medial points at scales ranging from 1 to 5.

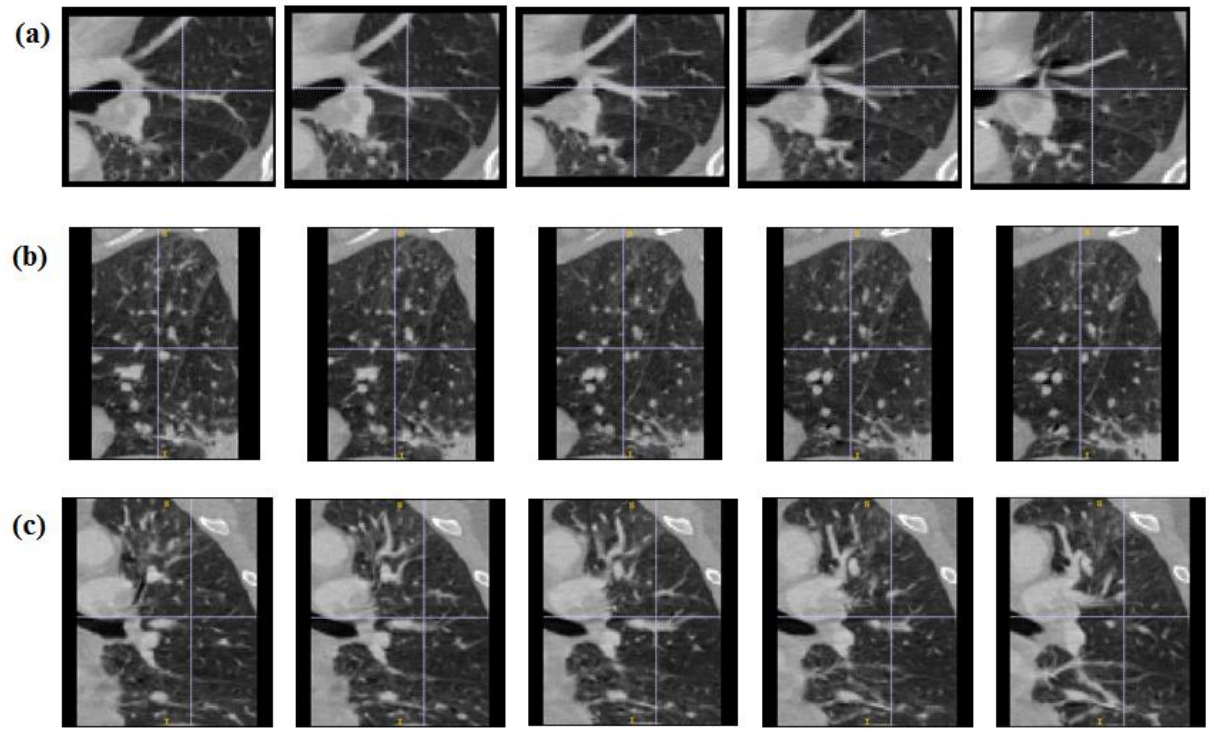

Figure 18: Five slices of the cropped and resampled 3D lung image used in our experiment, shown in the (a) Coronal, (b) Sagittal, And (c) Axial orientations.

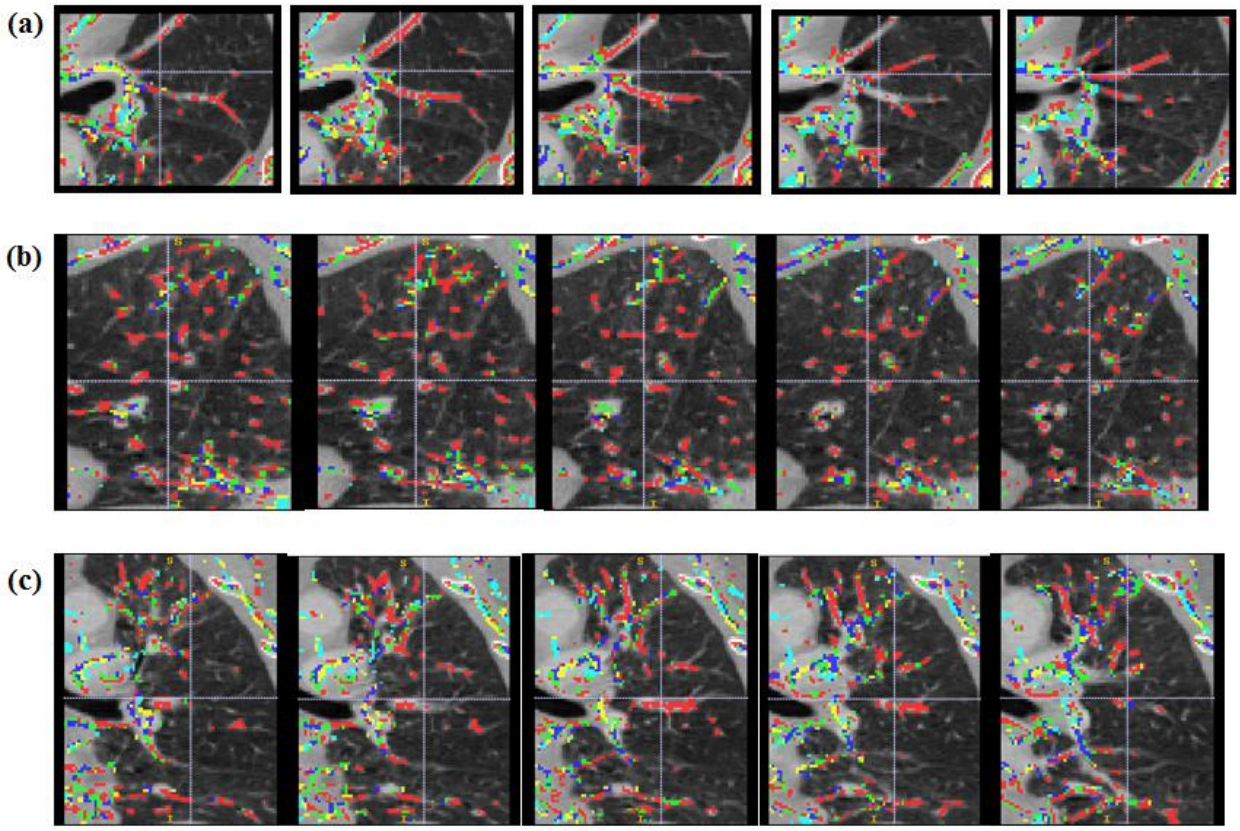

Figure 19: The slices shown in Figure 18 with the detected medial points superimposed. The scales at which the medial points are detected are color coded as follows: 1–red; 2–green; 3–dark blue; 4–yellow; 5–light blue.

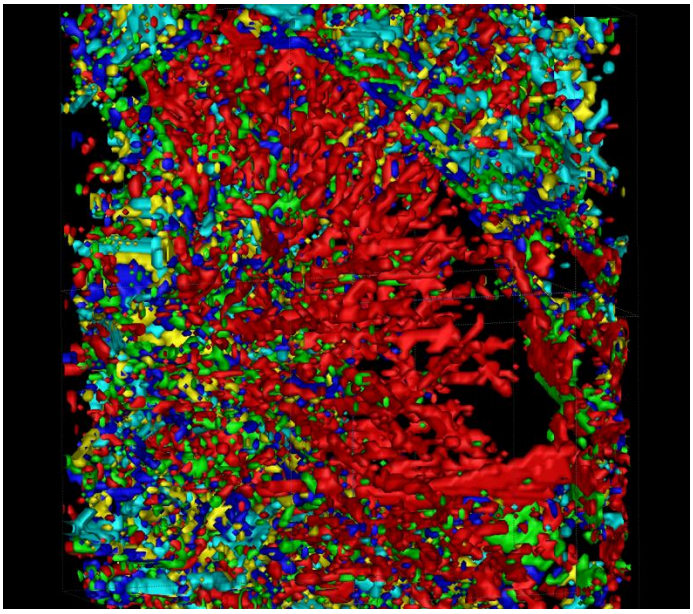

Figure 20: 3D rendering of the medial points using ITK-SNAP. Color coding as in Figure 19.

8.0 IMPLEMENTATION IN ITK

8.1 ABOUT ITK

ITK stands for *Insight Segmentation and Registration Toolkit*. It is an open-source, cross platform, object-oriented system used by developers for image analysis. It contains a collection of algorithms and functions mostly designed for medical image analysis. ITK was started in 1999 on the initiative of the National Library of Medicine (NLM) at the National Institutes of Health (NIH) [7]. As an open-source project, it has been created, debugged, maintained and extended by developers from around the world. ITK is implemented in C++ and is designed to run on many platforms. It can be downloaded for free from the ITK webpage: www.itk.org. ITK makes use of the CMake build environment to handle the compilation process. CMake, which stands for *cross-platform make* is a build environment that is operating system and compiler independent. It creates native makefiles and workspaces that can be used in many compiler environments [8]. Together, ITK and CMake provide researchers and developers a powerful means to implement their ideas and algorithms. ITK has a vast library that supports numerous image processing tasks, including image read/write, segmentation, registration, image transformations, interpolations, linear and nonlinear filtering, creating spatial objects, morphology, level sets etc.

ITK features a powerful plugin-based IO mechanism for reading and writing images. It supports a wide variety of image types such as bmp, analyze, DICOM, JPEG, MetaImage, png

etc. The output image in our case will consist of medial points in which pixel intensity corresponds to the scale at which the medial was detected.

8.2 IMPLEMENTATION

Figures 21, 22, and 23 show the organization of the various components, as will be discussed in this section.

8.2.1 Organization

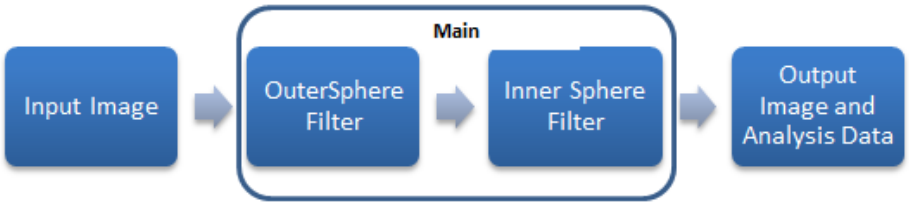

Figure 21: Input image is passed to an Inner sphere filter through an outer sphere filter to obtain the output image which consists of medial points and analysis data

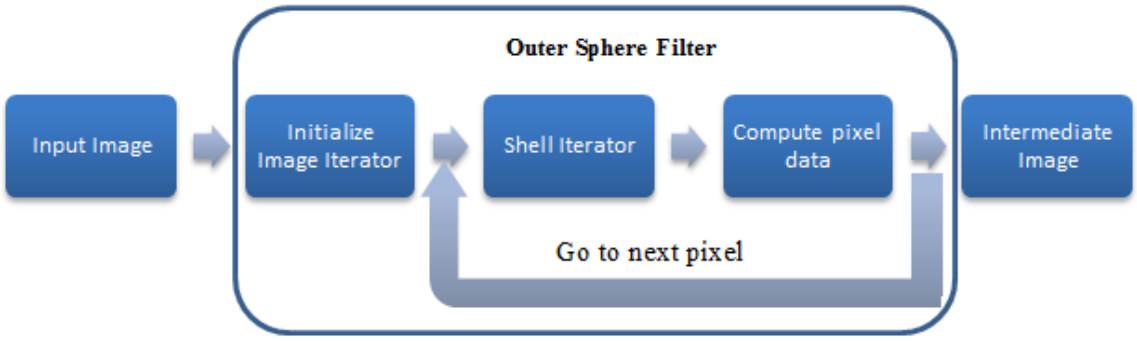

Figure 22: Outline of OuterSphereFilter

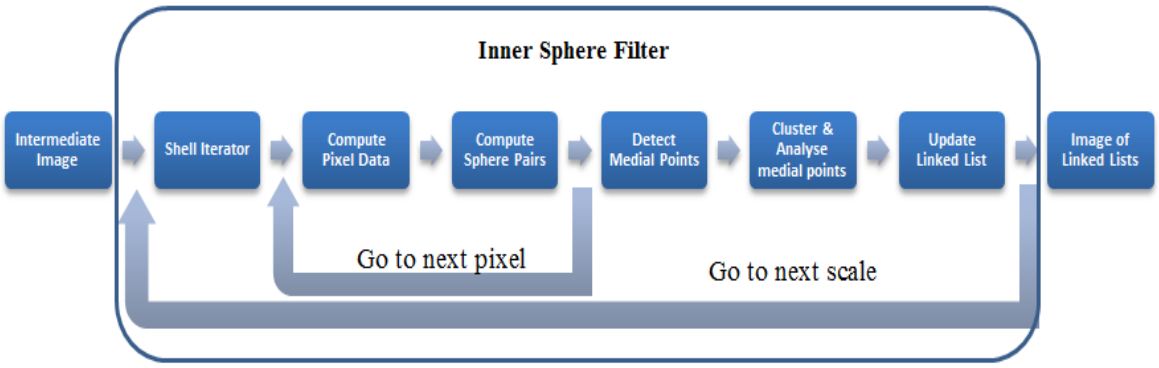

Figure 23: Outline of InnerSphereFilter

8.2.2 Iterators in ITK

The iterator in ITK is a powerful way to sequentially and efficiently access pixels in an image. There are several types of iterators used for specific purposes such as traversing image regions, local neighborhoods, arbitrary functions, random pixels etc. Iterators work in N-dimensions and are designed for computational efficiency. They are fundamental to our implementation.

8.2.3 Shell Iterator

Developed previously in our lab, the shell iterator is used to keep track of offsets from a central point. It is completely defined by offset values and is used to iterate through a shell. We use it to gather data to perform statistical calculations related to the outer and inner spheres and also to obtain the direction vector from the best sphere pair. The input parameters of the ITK function `itkShellIterator` are:

- Image on which the iterator operates

- Center location
- Size of the shell (scale)

8.2.4 Sphere Iterator

Similar to the shell iterator, this is used to iterate over a sphere neighborhood. The input parameters of `itkSphereIterator` are:

- Image on which the iterator operates
- Center location
- Size of the shell (scale)

8.2.5 Sphere Pixel Data

One of ITK's strengths is the ability to define "pixel" to mean just about anything. We custom define a class of pixels to hold data related to the spheres centered at each pixel. Each pixel instantiating an object of the Sphere Pixel Data class stores the following data:

Intensity Value – Stores the original image intensity

Shell Number – Keeps account of the scale

Pixel Count – Keeps count of the number of pixels in each sphere

Current Sum – Stores the sum of intensity values within a sphere. It is calculated as Current Sum + Next Intensity Value. It is updated with every pixel added to the sphere.

Current Sum Sq – Stores the square of Current Sum. It is calculated as Current Sum Sq + (Next Intensity Value ^ 2).

Outer Sphere Mean – Stores the outer sphere mean calculated in the outersphere filter.

Outer Sphere Variance – Stores the outer sphere variance calculated in the outersphere variance

Inner Sphere Mean – Stores the inner sphere mean calculated in the innersphere filter. It is updated whenever a new shell is added

Inner Sphere Variance – Stores the inner sphere variance calculated in the innersphere filter. It is updated whenever a new shell is added

Best Sphere Pair Test – Holds the *z* value of the optimum sphere pair.

Best Sphere Pair – Holds the vector of the best sphere pair

Best Unit Sphere Pair – Holds the unit vector of the best sphere pair

8.2.6 Linked List Pixel

This is a class that holds data related to each record in the linked list. Its members are:

The *z* value – Stores the *d'* of the sphere pair used to identify a medial point

Direction – Stores the unit vector of the sphere pair used to identify the medial point

Scale – Stores the scale at which the medial point was identified.

Eigenvalues – Stores the eigenvalues of a population of direction vectors at medial points at a given scale

8.2.7 Outer Sphere Filter

This filter takes as input the image on which the analysis needs to be performed (provided by the user). It makes use of the sphere iterator to compute the outer sphere at each pixel location. The

size of the outer sphere is set by the user. The filter produces an intermediate image as output, which stores statistical data such as mean and variance of the outer spheres.

8.2.8 Inner Sphere Filter

This filter takes as input the intermediate image generated by the outer sphere filter. The shell iterator is used to calculate statistical data related to the inner sphere. The minimum radius of the inner sphere is equal to the radius of the outer sphere. The maximum radius is set by the user. As mentioned before, a sphere can be computed as a union of shells. The shell iterator provides data from each shell, which is accumulated to form the sphere of the desired size. The Sphere Data Pixel is updated with the inner sphere values. Once the inner spheres are computed at all pixel locations, the medial points are detected by the procedure explained before. The Linked List Pixel is instantiated at the medial points and added to the Linked List at that location. The medial points are then clustered and analyzed using the procedure mentioned before. Eigenanalysis is performed on the clustered pixels and the eigenvalues are stored in a .txt file. This process is repeated at the next scale. When the maximum scale is reached, we obtain an image in which each pixel stores a linked list.

8.2.9 Eigenanalysis

Eigenanalysis has been implemented in ITK using itkCovarianceCalculator, which calculates the covariance matrix of the target sample data and itkSymmetricEigenAnalysis, which finds eigenvalues of a real 2D symmetric matrix.

9.0 CONCLUSION

In this thesis, I have developed a system that is capable of detecting medial points, which can be represented in scale space. Medial features such as scale, orientation, and dimensionality have been extracted and analyzed to detect basic local geometric shapes including, cylinder, slab, and sphere. Sphere pairs were computed throughout the image, and the one that generated the most significant *d'*, at a pixel location for a particular scale, was considered to be the optimum sphere pair. A mask, which forms pairs of pixels in each cardinal direction from a central pixel, was applied at each pixel location to compute the dot product of the direction vectors for each pair of pixels. Medial points were detected by applying a threshold to the dot product computed for each pair as well as a threshold to the minimum *d*' for the constituent sphere-pairs. This was repeated at higher scales by increasing the radius of the inner sphere. Clusters of the medial points were formed by describing a local region within the inner sphere of each medial point at its particular scale, and the corresponding direction vectors were subjected to eigenanalysis to identify the local medial dimensionality and orientation. Tests were performed on three synthetic 3D objects: a torus, a hollow sphere, and a sphere. The system detected the medial manifolds of the three objects accurately even in the presence of noise, and the local shapes of the objects were identified successfully in the case of the torus and hollow sphere. A method for extending this to the sphere was proposed. The results confirmed the capability of the system to detect medial points at more than one scale and to identify the described medial shape features effectively.

10.0 FUTURE WORK

Our goal is to demonstrate the system's capability for detecting medial manifolds in medical images such as MRI and CT in a useful manner. To accomplish this, we are considering a number of improvements.

As already mentioned in Section 5.0, the criteria used to cluster the direction vectors may be altered to include the direction vectors from sphere pairs with lower *d'* (yet significant). This would result in denser clusters around the center of the sphere, thus aiding the eigenanalysis of particularly focused local shapes such as the sphere.

Many steps in the methods described contain parameters that need to be optimized in terms of the overall accuracy of the system. The C++ code can also be optimized to minimize speed and memory.

Beyond this, we intend to extend the shape detection methodology to detect more complex shapes present in anatomical structures. Initial targets include extended cylinders, branching cylinders, and cylinders that pass by each other. By following a basic "bottom-up" approach to increasingly complex structures within larger regions, our ultimate goal is to provide a new framework for the description and analysis of shape within medical images.

APPENDIX

ITK CODE

Main

```
/** Includes */
#include "itkShellIterator.h"
#include "mainheader.h"

//Main function
int main()
{
      try
      {
            // Maximum size of the inner and outer spheres
            int maxOuterSphereSize = 1;
            int maxInnerSphereSize = 10;

            std::cout<<"Running the pipeline......."<<std::endl;

            //read the input image
            ReaderType::Pointer reader = ReaderType::New();
            Reader->SetFileName ((2==MY_DIMENSION)?
            "Bronchi_2d.mha":"Torus_15_6.mha"  );
            CastFilterTypetoRead::Pointer castfiltertoread =
            CastFilterTypetoRead::New();
            castfiltertoread->SetInput(reader->GetOutput());

            // Running the outer sphere filter
            m_OuterSphereFilter = OuterSphereFilterType::New();
            m_OuterSphereFilter->SetInput(castfiltertoread->GetOutput());
            m_OuterSphereFilter->SetOuterSphereSize(maxOuterSphereSize);

            // Running the inner sphere filter
            //Set the minimum and maximum radius as set by the user
            m_InnerSphereFilter = InnerSphereFilterType::New();
            m_InnerSphereFilter->SetInput(m_OuterSphereFilter->GetOutput());
            m_InnerSphereFilter->SetMaxInnerSphereSize(maxInnerSphereSize);
            m_InnerSphereFilter->SetMinInnerSphereSize(maxOuterSphereSize);
            m_InnerSphereFilter->Update();

            m_ResultImage = castfiltertoread->GetOutput();
            //Get the output of the innerspherefilter
            m_LinkedListImage = m_InnerSphereFilter->GetOutput();
```

```
//Convert the result into a image viewable by the user
//Pixel intensity corresponds to scale at which the medial was
found

typedef itk::ImageRegionIterator<LinkedListImageType>
LinkedListItType; // Iterator type
LinkedListItType LinkedListIt(m_LinkedListImage,
m_LinkedListImage->GetRequestedRegion());

ItType resultIt(m_ResultImage, m_ResultImage-
>GetRequestedRegion());

LinkedListIt.GoToBegin();
resultIt.GoToBegin();

std::list<LinkedListPixel<MY_DIMENSION>> TempList;
std::list<LinkedListPixel<MY_DIMENSION>>::iterator TempListIt;

//Produce a text file containing indormation related to medial
points
std::ofstream LLfile ("LinkedListDataTorus156_10_27_A.txt");
if (LLfile.is_open())
LLfile<<"Index[0]\t[1]\t[2] \tScale\tDirection[0]
\t[1]\t[2]\tZValue\n";

double LLScale;
itk::Vector <double, 3> LLDirection;
double LLZValue;

typedef itk::FixedArray< double, MY_DIMENSION >
EigenValuesArrayType;
EigenValuesArrayType EigenValues;

while (!LinkedListIt.IsAtEnd())
{
      resultIt.Set(0); //Set the result image to 0 everywhere
      other than at the medial

      TempList = LinkedListIt.Get();
      TempListIt = TempList.begin();

      while(TempListIt != TempList.end())
      {
            LLScale = TempListIt->getScale();
            LLDirection = TempListIt->getDirection();
            LLZValue = TempListIt->getZValue();
            EigenValues = TempListIt->getEigenValues();
            if (LLfile.is_open())
            {
                  LLfile<<temporaryIndex[0]
                  <<"\t"<<temporaryIndex[1] <<"\t"
                  <<temporaryIndex[2] <<"\t"<<LLScale<<"\t"
                  <<LLDirection[0] <<"\t"<<LLDirection[1]
                  <<"\t"<<LLDirection[2]<<"\t"<<LLZValue<<"\n";
            }

            resultIt.Set(TempListIt->getScale());
```

```
                        TempListIt++;
                }

                LinkedListIt++;
                resultIt++;

        }
        if (LLfile.is_open())
        {
                LLfile.close();
        }

        SaveImage(m_ResultImage,"Torus156Testing_10_27_A.mha" );

        return 0;
    }


    catch( itk::ExceptionObject & excp )
  {
        std::cerr << "Exception caught: " << std::endl;
        std::cerr << excp << std::endl;
        std::cin.get();
        return EXIT_FAILURE;
  }
}
```

Mainheader.h

```
#define MY_DIMENSION 3

/** Standard includes */
#include <iostream>
#include "math.h"

/** Includes the ITK header files */
#include "itkImage.h"
#include "itkImageRegionIterator.h"
#include "itkImageFileWriter.h"
#include "itkImportImageFilter.h"
#include "itkImageFileReader.h"
#include "itkMeanImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
#include "itkCastImageFilter.h"

// Shells and Spheres includes
#include "itkShellIterator.h"
#include "itkSphereIterator.h"
#include "itkOuterSphereFilter.h"
#include "itkInnerSphereFilter.h"
#include "itkSphereDivergenceFilter.h"
#include <math.h>
#include <list>

//#include "LinkedListPixel.h"

#define PI 3.14159265

/** ITK typedefs */
typedef itk::Image< double, MY_DIMENSION > ImageType;
typedef itk::Image< SphereDataPixel<MY_DIMENSION>, MY_DIMENSION >
SphereImageType;
#if MY_DIMENSION == 2
  typedef itk::Image< double, MY_DIMENSION > WriteImageType;
#else
  typedef itk::Image<  short, MY_DIMENSION > WriteImageType;
#endif
// Iterator type
typedef itk::ImageRegionIterator<ImageType> ItType;
typedef itk::ImageRegionIterator<SphereImageType> SphereItType;
// Our writer for saving the image
typedef itk::ImageFileWriter< ImageType > WriterType;
typedef itk::ImageFileReader< WriteImageType > ReaderType;
// The mean filter was used since it is one of the simplest blurring filters
typedef itk::MeanImageFilter< ImageType, ImageType >  MeanFilterType;
// Our filter to do outer spheres
typedef itk::OuterSphereFilter< ImageType, MY_DIMENSION >
OuterSphereFilterType;
// Rescaler to save
typedef itk::RescaleIntensityImageFilter< ImageType, WriteImageType >
RescaleFilterType;
typedef itk::RescaleIntensityImageFilter< WriteImageType, ImageType >
LoaderRescaleFilterType;
typedef itk::InnerSphereFilter< MY_DIMENSION > InnerSphereFilterType;
typedef itk::SphereDivergenceFilter< MY_DIMENSION >
SphereDivergenceFilterType;
```

```
typedef itk::CastImageFilter< ImageType, WriteImageType >
CastFilterTypetoWrite;
typedef itk::CastImageFilter< WriteImageType, ImageType >
CastFilterTypetoRead;


typedef itk::Image<std::list<LinkedListPixel<MY_DIMENSION>>, MY_DIMENSION >
LinkedListImageType;


ImageType::Pointer m_Image;
SphereImageType::Pointer m_OuterSphere;
ImageType::Pointer m_ResultImage;
SphereImageType::Pointer m_SpherePair;
MeanFilterType::Pointer m_MeanFilter;
OuterSphereFilterType::Pointer m_OuterSphereFilter;
InnerSphereFilterType::Pointer m_InnerSphereFilter;
SphereDivergenceFilterType::Pointer m_SphereDivergenceFilter;

LinkedListImageType::Pointer m_LinkedListImage;

// Please limit the filename to below 100 letters
bool SaveImage(ImageType::Pointer m_ResultImage,char filename[100])
{

      // Setting up the writer
      WriterType::Pointer writer = WriterType::New();
      writer->SetFileName(filename); // The filename we want to save as
      writer->SetInput(m_ResultImage); // The image that is saved

      // Error checking code from the ITK example
      try
      {
            writer->Update();
      }
      catch( itk::ExceptionObject & err )
      {
            std::cerr << "ExceptionObject caught !" << std::endl;
            std::cerr << err << std::endl;
            std::cin.get();
            return EXIT_FAILURE;
      }
}
```

SphereDataPixel.h

```
// Class of pixels holding information about the spheres
// Now templated for multiple dimensions!

template <unsigned int VImageDimension> class SphereDataPixel
{
public:
      double intensityValue; // Keep the original image intensity
      int shellNumber;//Keeps track to what shell we have already iterated to
      int pixelCount; // Total numer of pixels that we have iterated through
      double outerSphereMean; // Calculated in the outer sphere filter
      double outerSphereVariance; // Calculated in the outer sphere filter
      double innerSphereMean; // Constantly updated as another shell is added
      to the sphere, currentSum / pixelCount
      double innerSphereVariance; // (currentSumSq - (currentSum *
      innerSphereMean))/(pixelCount - 1)
      double currentSum; // currentSum + next value
      double currentSumSq; // currentSumSq + (next value ^ 2)

      itk::Offset <VImageDimension> bestSpherePair; // Holds the vector to
      the best sphere pair
      itk::Vector <double, VImageDimension> bestUnitSpherePair; // Holds the
      unit vector to the best sphere pair
      // The partial refers to df/dx, df/dy and df/dz
      itk::Vector <double, VImageDimension> partialDerivatives; // [0] is x,
      [1] is y, [2] is z and so on...
      double bestSpherePairTest;
      double currentSpherePairTest;
      double divergence;
      void InitializeSphereData( double value )
      {
            intensityValue = value;
            shellNumber = 0;
            pixelCount = 0;
            outerSphereMean = 0;
            outerSphereVariance = 0;
            innerSphereMean = 0;
            innerSphereVariance = 0;
            currentSum = 0;
            currentSumSq = 0;
            bestSpherePairTest = 0;
            currentSpherePairTest = 0;
            divergence = 0;
            LowIntensityPixels = 0;
            HighIntensityPixels = 0;

            for (unsigned int counter = 0; counter < VImageDimension;
            counter++)
            {
                  bestSpherePair[counter] = 0;
                  bestUnitSpherePair[counter] = 0;
                  partialDerivatives[counter] = 0;
            }
      }
};
```

LinkedListPixel.h

```
template <unsigned int VImageDimension> class LinkedListPixel
{
public:
	double ZValue;
	itk::Vector <double, VImageDimension> Direction; // Holds the unit
vector
	//to the best sphere pair
	int Scale;
	typedef itk::FixedArray< double, VImageDimension >
EigenValuesArrayType;
	EigenValuesArrayType EigenValues;



	void InitializeLinkedList()
	{
		Scale = 0;
		ZValue = 0;
		for (unsigned int counter = 0; counter < VImageDimension;
counter++)
		{
			Direction[counter] = 0;
		}
		for (unsigned int counter = 0; counter < VImageDimension;
counter++)
		{
			EigenValues[counter] = 0;
		}

	}

	int getScale()
	{
		return Scale;
	}

	itk::Vector <double, VImageDimension> getDirection()
	{
		return Direction;
	}

	double getZValue()
	{
		return ZValue;
	}

	EigenValuesArrayType getEigenValues()
	{
		return EigenValues;
	}

};
```

Itkshelliterator

```
#ifndef __itkShellIterator_h
#define __itkShellIterator_h

#include "itkImageIterator.h"
#include "itkOffset.h"

#include <list>
#include <vector>

namespace itk
{

/**
 * \class ShellIterator
 * \brief Iterates over a sphere shell.
 *
 *  Here we have an iterator that keeps track of offsets from a central
point.
 *  The iterator is totally defined by offset values.  We keep these as a
list
 *  of indices.  We also need a function to set the center of the iterator,
 *  so we can calculate the included indices based on it.  Default center
will
 *  origin ( (0,0,0) for 3D)
 *
 *  We have a list of indices representing offsets.
 *  center + offset = index => offset = index - center
 *
 *
 * \ingroup ImageIterators
 *
 */
template <typename TImage>
class ITK_EXPORT ShellIterator : public ImageIterator<TImage>
      //ITK_EXPORT?
{
public:
  /** Standard class typedefs. */
  typedef ShellIterator Self;
  typedef ImageIterator<TImage> Superclass;

  /** Number of dimensions */
  itkStaticConstMacro(NDimensions, unsigned int, TImage::ImageDimension);
      //itkStaticConstMacro?

  /** Index typedef support. */
  typedef typename Superclass::IndexType  IndexType;

  /** Size typedef support. */
  typedef typename Superclass::SizeType    SizeType;

  /** Region typedef support */
  typedef typename Superclass::RegionType    RegionType;

  /** Image typedef support. */
  typedef typename Superclass::ImageType   ImageType;
```

```
  /** Internal Pixel Type */
  typedef typename Superclass::InternalPixelType   InternalPixelType;
  /** External Pixel Type */
  typedef typename Superclass::PixelType   PixelType;
  typedef Offset<NDimensions> OffsetType;

  /** An stl storage container type that can be sorted.  The type used for
      the list of active offsets in the neighborhood.*/
  typedef std::list<OffsetType> OffsetListType;
  typedef OffsetListType* OffsetListPointerType;
  typedef typename OffsetListType::iterator OffsetListIteratorType;

  /** std::list containing the vectors to each of the points in the outer
shell.*/
  typedef vnl_vector<double> VectorType;
  typedef VectorType* VectorPointerType;
  typedef std::list<VectorPointerType> VectorListType;
  typedef VectorListType* VectorListPointerType;
  typedef VectorListType::iterator VectorListIteratorType;

  /** Constructor establishes an iterator to walk a particular image */
  ShellIterator(ImageType *imagePtr, const IndexType& center, int scale, bool
ioi = false);

  /** Default Destructor. */
  ~ShellIterator() {}
  /** Compute whether the index of interest is a legal index of the image */
  bool IsIndexWithinImage() const;

  /** Walk forward one index */
  void operator++() {
            Next();
  }
  void Next() {
            do {
                  ++m_OffsetListIterator;
            } while( ( m_OffsetListIterator != m_OffsetList->end() ) &&
                        !m_IncludeOutsideIndices && !IsIndexWithinImage() );
  }
  bool IsAtEnd() {
    if(m_OffsetListIterator == m_OffsetList->end()) return true;
    else return false;
  }

  void GoToBegin() {
            m_OffsetListIterator = m_OffsetList->begin();
            while( !m_IncludeOutsideIndices && !IsAtEnd() &&
!IsIndexWithinImage() )
            {
                  ++m_OffsetListIterator;
            }
      }

  /** Get the pixel value */
  const PixelType & Get(void) const
    {
      // This will give the index of the center pixel + offset
```

```
      IndexType index = m_CenterIndex + *m_OffsetListIterator;
        PixelType myPixel;
        myPixel = this->m_Image->GetPixel( index ); // It can't access the
pixel correctly...WHY???
      return this->m_Image->GetPixel( index );
    }
      /** non-const Get */
      PixelType& Value()
      {
    IndexType index = m_CenterIndex + *m_OffsetListIterator;
    return const_cast<ImageType*>((const ImageType*)this->m_Image)->GetPixel(
index );
      }
  /** Get the offset. This provides a read only reference to the offset.
   * This causes the offset to be calculated from pointer arithmetic and is
   * therefore an expensive operation.*/
  const OffsetType GetOffset()
    { return *m_OffsetListIterator; }
/** Get the index. This provides a read only reference to the index.
   * This causes the index to be calculated from pointer arithmetic and is
   * therefore an expensive operation.
   * \sa SetIndex */
  const IndexType GetIndex()
    {
      // This will give the index of the center pixel + offset
      IndexType index = m_CenterIndex + *m_OffsetListIterator;
      return index;
    }
      Self& operator=( const Self& source )
      {
            *(Superclass*)this = (const Superclass&)source;
            m_CenterIndex = source.m_CenterIndex;
            m_OffsetList = source.m_OffsetList;
            m_OffsetListIterator = source.m_OffsetListIterator;
            return (*this);
      }
protected: //made protected so other iterators can access
  IndexType m_CenterIndex;
  OffsetListPointerType m_OffsetList;
  OffsetListIteratorType m_OffsetListIterator;

  VectorListPointerType m_VectorList;
  VectorListIteratorType m_VectorListIterator;

      // static vector of shell offset lists, indexed by scale
      static std::vector<OffsetListPointerType> s_ShellOffsetLists;
      static int s_HighestGeneratedScale;
      bool m_IncludeOutsideIndices;
};

} // end namespace itk

#ifndef ITK_MANUAL_INSTANTIATION
#include "itkShellIterator.txx"
#endif

#endif
```

itkShellIterator.txx

```
#ifndef _itkShellIterator_txx
#define _itkShellIterator_txx

#include <iostream>
#include "itkShellIterator.h"
#include "vnl/vnl_vector_fixed.h"

namespace itk
{

template<class TImage>
std::vector<typename ShellIterator<TImage>::OffsetListPointerType>
ShellIterator<TImage>::s_ShellOffsetLists;

template <class TImage>
int ShellIterator<TImage>::s_HighestGeneratedScale = -1;

template<class TImage>
ShellIterator<TImage>
::ShellIterator(ImageType *imagePtr, const IndexType& center, int scale, bool
ioi): ImageIterator<TImage>( imagePtr, imagePtr->GetLargestPossibleRegion() )
{
      m_CenterIndex = center;
      m_IncludeOutsideIndices = ioi;

      // Generate new offset lists if we need to
      if( scale > s_HighestGeneratedScale ) {
            for( int s = s_HighestGeneratedScale+1; s <= scale; ++s ) {
                  OffsetListPointerType new_list = new OffsetListType();
                  s_ShellOffsetLists.push_back( new_list );
                  OffsetType current_offset;
                  current_offset.Fill(-s);
// Inefficient, but practically it won't make a significant difference.
                  while( 1 ) {
// compute nearest-integer distance to the center. if equal to this scale,
include
// the pixel in this scale's offset list
                        vnl_vector_fixed<double,NDimensions> v_offset;
                        for( int i = 0; i < NDimensions; ++i ) v_offset[i] =
double( current_offset[i] );
                        if( int( v_offset.magnitude() + 0.5 ) == s ) {
                              new_list->push_back( current_offset );
                        }
                        // move onto the next pixel
                        int i;
                        for( i = 0; i < NDimensions; ++i ) {
                              ++current_offset[i];
                              if( current_offset[i] > s ) current_offset[i] =
-s;
                              else break;
                        }
// every component overflowed past scale, so we've hit every pixel. done.
                        if( i == NDimensions ) break;
                  }
            }
            s_HighestGeneratedScale = scale;
      }
```

```
  m_OffsetList = s_ShellOffsetLists[scale];
}

template<class TImage>
bool
ShellIterator<TImage>
::IsIndexWithinImage() const
{
  SizeType size = this->m_Region.GetSize();
      OffsetType& offset = *m_OffsetListIterator;
      IndexType index = m_CenterIndex + offset;
  for(int i = 0;i < NDimensions; ++i) {
    if(index[i] >= size[i] || index[i] < 0) return false;
  }
  return true;
}

} // end namespace itk

#endif
```

itkSphereIterator.h

```
#ifndef __itkSphereIterator_h
#define __itkSphereIterator_h

#include "itkImageIterator.h"
#include "itkShellIterator.h"
#include "itkOffset.h"

#include <list>
#include <vector>

namespace itk
{

/**
 * \class SphereIterator
 * \brief Iterates over a sphere neighborhood.
 *
 * \ingroup ImageIterators
 *
 */
template <typename TImage>
class ITK_EXPORT SphereIterator : public ImageIterator<TImage>
{
public:
  /** Standard class typedefs. */
  typedef SphereIterator Self;
  typedef ImageIterator<TImage> Superclass;

  /** Number of dimensions */
  itkStaticConstMacro(NDimensions, unsigned int, TImage::ImageDimension);

  /** Index typedef support. */
  typedef typename Superclass::IndexType  IndexType;

  /** Size typedef support. */
  typedef typename Superclass::SizeType    SizeType;

  /** Region typedef support */
  typedef typename Superclass::RegionType    RegionType;

  /** Image typedef support. */
  typedef typename Superclass::ImageType   ImageType;

  /** Internal Pixel Type */
  typedef typename Superclass::InternalPixelType   InternalPixelType;

  /** External Pixel Type */
  typedef typename Superclass::PixelType   PixelType;

      typedef Offset<NDimensions> OffsetType;

  /** An stl storage container type that can be sorted.  The type used for
      the list of active offsets in the neighborhood.*/
  typedef std::list<OffsetType> OffsetListType;
  typedef OffsetListType* OffsetListPointerType;
  typedef typename OffsetListType::iterator OffsetListIteratorType;
```

```
/** std::list containing the vectors to each of the points in the outer
shell.*/
  typedef vnl_vector<double> VectorType;
  typedef VectorType* VectorPointerType;
  typedef std::list<VectorPointerType> VectorListType;
  typedef VectorListType* VectorListPointerType;
  typedef VectorListType::iterator VectorListIteratorType;

  /** Constructor establishes an iterator to walk a particular image */
  SphereIterator(ImageType *imagePtr, const IndexType& center, int scale,
bool ioi = false);

  /** Default Destructor. */
  ~SphereIterator() {}

  /** Compute whether the index of interest is a legal index of the image */
  bool IsIndexWithinImage() const {
            return m_CurrentShellIterator.IsIndexWithinImage();
      }

  /** Walk forward one index */
  void operator++() {
            Next();
  }

  void Next() {
            ++m_CurrentShellIterator;
            if( m_CurrentShellIterator.IsAtEnd() ) {
                  ++m_CurrentScale;
                  if( m_CurrentScale <= m_Scale ) {
                        m_CurrentShellIterator = ShellIterator<ImageType>(
                                    const_cast<ImageType*>((const
ImageType*)this->m_Image), m_CenterIndex,
                                    m_CurrentScale );
                        m_CurrentShellIterator.GoToBegin();
                  }
            }
      }

  bool IsAtEnd() {
            return ( (m_CurrentScale >= m_Scale) &&
m_CurrentShellIterator.IsAtEnd() );
  }

  void GoToBegin() {
            m_CurrentScale = 0;
            m_CurrentShellIterator = ShellIterator<ImageType>(
                        const_cast<ImageType*>((const ImageType*)this-
>m_Image), m_CenterIndex, 0 );
            m_CurrentShellIterator.GoToBegin();
      }

  /** Get the pixel value */
  const PixelType & Get(void) const {
      return m_CurrentShellIterator.Get();
    }
```

```
      /** non-const Get */
      PixelType& Value() {
      return m_CurrentShellIterator.Value();
      }

  /** Get the offset. This provides a read only reference to the offset.
   * This causes the offset to be calculated from pointer arithmetic and is
   * therefore an expensive operation.*/
  const OffsetType GetOffset()
    { return m_CurrentShellIterator.GetOffset(); }

/** Get the index. This provides a read only reference to the index.
   * This causes the index to be calculated from pointer arithmetic and is
   * therefore an expensive operation.
   * \sa SetIndex */
  const IndexType GetIndex()
    {
      return m_CurrentShellIterator.GetIndex();
    }

      Self& operator=( const Self& source )
      {
            *(Superclass*)this = (const Superclass&)source;
            m_CenterIndex = source.m_CenterIndex;
            m_Scale = source.m_Scale;
            m_CurrentScale = source.m_CurrentScale;
            //m_CurrentShell = source.m_CurrentShell;
      }

protected: //made protected so other iterators can access
  IndexType m_CenterIndex;

      int m_Scale;

      int m_CurrentScale;
      ShellIterator<ImageType> m_CurrentShellIterator;

      bool m_IncludeOutsideIndices;
};

} // end namespace itk

#ifndef ITK_MANUAL_INSTANTIATION
#include "itkSphereIterator.txx"
#endif

#endif
```

itkSphereIterator.txx

```
#ifndef _itkShellIterator_txx
#define _itkShellIterator_txx

#include <iostream>
#include "itkShellIterator.h"
#include "vnl/vnl_vector_fixed.h"

namespace itk
{

template<class TImage>
std::vector<typename ShellIterator<TImage>::OffsetListPointerType>
ShellIterator<TImage>
::s_ShellOffsetLists;

template <class TImage>
int
ShellIterator<TImage>
::s_HighestGeneratedScale = -1;


template<class TImage>
ShellIterator<TImage>
::ShellIterator(ImageType *imagePtr, const IndexType& center, int scale, bool
ioi)
      : ImageIterator<TImage>( imagePtr, imagePtr->GetLargestPossibleRegion()
)
{
      m_CenterIndex = center;
      m_IncludeOutsideIndices = ioi;

      // Generate new offset lists if we need to
      if( scale > s_HighestGeneratedScale ) {
            for( int s = s_HighestGeneratedScale+1; s <= scale; ++s ) {
                  OffsetListPointerType new_list = new OffsetListType();
                  s_ShellOffsetLists.push_back( new_list );
                  OffsetType current_offset;
                  current_offset.Fill(-s);
                  // Inefficient, but practically it won't make a significant
                  difference.
                  while( 1 ) {
                  // compute nearest-integer distance to the center. if equal
                  to this  //scale, include
                        // the pixel in this scale's offset list
                        vnl_vector_fixed<double,NDimensions> v_offset;
                        for( int i = 0; i < NDimensions; ++i ) v_offset[i] =
                        double( current_offset[i] );
                        if( int( v_offset.magnitude() + 0.5 ) == s ) {
                              new_list->push_back( current_offset );
                        }
                        // move onto the next pixel
                        int i;
                        for( i = 0; i < NDimensions; ++i ) {
                              ++current_offset[i];
                              if( current_offset[i] > s ) current_offset[i] =
                              -s;
                              else break;
```

```
                        }
                        // every component overflowed past scale, so we've
                        hit every pixel. done.
                        if( i == NDimensions ) break;
                    }
                }
                s_HighestGeneratedScale = scale;
        }

  m_OffsetList = s_ShellOffsetLists[scale];
}

template<class TImage>
bool
ShellIterator<TImage>
::IsIndexWithinImage() const
{
  SizeType size = this->m_Region.GetSize();
      OffsetType& offset = *m_OffsetListIterator;
      IndexType index = m_CenterIndex + offset;
  for(int i = 0;i < NDimensions; ++i) {
    if(index[i] >= size[i] || index[i] < 0) return false;
  }
  return true;
}

} // end namespace itk

#endif
```

itkOuterSphereFilter.h

```
#ifndef __itkOuterSphereFilter_h
#define __itkOuterSphereFilter_h

#include "itkImageToImageFilter.h"

// Shells and spheres specific includes
#include "itkShellIterator.h"
#include "itkSphereIterator.h"
#include "SphereDataPixel.h"

#include <math.h>

namespace itk
{

/** \class OuterSphereFilter
 * \Calculates the mean and standard deviation for outer spheres
 *
 * This class is parameterized over the type of the input image and
 * the type of the output image.
 *
 * \ingroup
 */
template <class TInputImage, unsigned int VImageDimension>
class ITK_EXPORT OuterSphereFilter : public ImageToImageFilter<TInputImage,
itk::Image< SphereDataPixel<VImageDimension>, VImageDimension> >
{
public:
  /** Standard class typedefs. */
  typedef OuterSphereFilter                                  Self;
  typedef ImageToImageFilter<TInputImage,itk::Image<
SphereDataPixel<VImageDimension>, VImageDimension > >  Superclass;
  typedef SmartPointer<Self>                                 Pointer;
  typedef SmartPointer<const Self>                           ConstPointer;

  /** Method for creation through the object factory. */
  itkNewMacro(Self);

  /** Run-time type information (and related methods). */
  itkTypeMacro(OuterSphereFilter, ImageToImageFilter);

  /** Some convenient typedefs. */
  typedef TInputImage                              InputImageType;
  typedef typename    InputImageType::Pointer    InputImagePointer;
  typedef typename    InputImageType::RegionType InputImageRegionType;
  typedef typename    InputImageType::PixelType  InputImagePixelType;

  typedef itk::Image< SphereDataPixel<VImageDimension>, VImageDimension>
OutputImageType;
  typedef typename     OutputImageType::Pointer    OutputImagePointer;
  typedef typename     OutputImageType::RegionType OutputImageRegionType;
  typedef typename     OutputImageType::PixelType  OutputImagePixelType;
  typedef typename     OutputImageType::IndexType  OutputImageIndexType;

  /** Set the direction in which to reflect the data. */
  itkGetConstMacro( Direction, unsigned int );
  itkSetMacro( Direction, unsigned int );
```

```
  itkSetMacro( OuterSphereSize, int );

  /** ImageDimension constants */
  itkStaticConstMacro(InputImageDimension, unsigned int,
                      TInputImage::ImageDimension);
  itkStaticConstMacro(OutputImageDimension, unsigned int,
                      OutputImageType::ImageDimension);

#ifdef ITK_USE_CONCEPT_CHECKING
  /** Begin concept checking */
  itkConceptMacro(SameDimensionCheck,
    (Concept::SameDimension<InputImageDimension, OutputImageDimension>));
  /** End concept checking */
#endif

protected:
  OuterSphereFilter();
  virtual ~OuterSphereFilter() {};
  void PrintSelf(std::ostream& os, Indent indent) const;

  /** This method implements the actual reflection of the image.
   *
   * \sa ImageToImageFilter::ThreadedGenerateData(),
   *     ImageToImageFilter::GenerateData()  */
  void GenerateData(void);

private:
  OuterSphereFilter(const Self&); //purposely not implemented
  void operator=(const Self&); //purposely not implemented

  unsigned int m_Direction; // Not currently used
  int m_OuterSphereSize; // Maximum size of the outer spheres

};

} // end namespace itk

#ifndef ITK_MANUAL_INSTANTIATION
#include "itkOuterSphereFilter.txx"
#endif

#endif
```

itkOuterSphereFilter.txx

```
#ifndef __itkOuterSphereFilter_txx
#define __itkOuterSphereFilter_txx

#include "itkOuterSphereFilter.h"
#include "itkImageLinearIteratorWithIndex.h"
#include "itkImageLinearConstIteratorWithIndex.h"
#include "itkProgressReporter.h"

namespace itk
{

	/**
	* Constructor
	*/
	template <class TInputImage, unsigned int VImageDimension>
	OuterSphereFilter<TInputImage,VImageDimension >
		::OuterSphereFilter()
	{
		this->SetNumberOfRequiredInputs( 1 ); // We are only taking in 1
		input, the image dimension
		m_Direction = 0;
		m_OuterSphereSize = 0;
	}


	/**
	* GenerateData goes through every pixel and calculates the mean at the
	pixel
	*/
	template <class TInputImage, unsigned int VImageDimension>
	void OuterSphereFilter<TInputImage,VImageDimension>
		::GenerateData( void )
	{
		std::cout<<std::endl<<"Running OuterSphere Filter";
		// Input image pointer and output image pointer
		typename Superclass::InputImageConstPointer inputPtr = this-
		>GetInput();
		typename Superclass::OutputImagePointer outputPtr = this-
		>GetOutput(0);

		// Making the output the same size as the input...
		outputPtr->SetRequestedRegion( inputPtr->GetRequestedRegion() );
		outputPtr->SetBufferedRegion( inputPtr->GetBufferedRegion() );
		outputPtr->SetLargestPossibleRegion( inputPtr-
		>GetLargestPossibleRegion() );
		outputPtr->Allocate();

		// Creating our iterators to go through the output/input
		typedef ImageRegionIterator<TInputImage> InputIterator;
		typedef ImageRegionIterator<OutputImageType> OutputIterator;
		InputIterator inputIt( const_cast<InputImageType*>(this-
		>GetInput()),  inputPtr->GetRequestedRegion() );
		OutputIterator outputIt( outputPtr, outputPtr-
		>GetRequestedRegion() );

		// Starting both input and output
		inputIt.GoToBegin();
```

```
	outputIt.GoToBegin();

	// Temporary variables to store the current sphere data and the
	current pixel value
	SphereDataPixel<VImageDimension> currentSphereDataPixel;
	double currentValue;
	// itk::Offset<VImageDimension> myOffset;
	while( !inputIt.IsAtEnd() )
	{
		// Sets all the sphere data to 0
		currentSphereDataPixel.InitializeSphereData( double
		(inputIt.Get()) );

		itk::SphereIterator <TInputImage>
		sphereIt(const_cast<InputImageType*>(this->GetInput()),
		inputIt.GetIndex(), m_OuterSphereSize, true);
		for(sphereIt.GoToBegin(); !sphereIt.IsAtEnd(); ++sphereIt)
		{
			if(sphereIt.IsIndexWithinImage()) // Edge pixels?
			{
				// myOffset = sphereIt.GetOffset();
				currentValue = double(sphereIt.Get());
				currentSphereDataPixel.currentSum =
				currentValue +
				currentSphereDataPixel.currentSum;
				currentSphereDataPixel.currentSumSq =
				currentSphereDataPixel.currentSumSq +
				(currentValue * currentValue);
				++currentSphereDataPixel.pixelCount;
			}
		}

		currentSphereDataPixel.outerSphereMean =
		currentSphereDataPixel.currentSum /
		currentSphereDataPixel.pixelCount;
		currentSphereDataPixel.outerSphereVariance =
		(currentSphereDataPixel.currentSumSq +
		(currentSphereDataPixel.outerSphereMean *
		currentSphereDataPixel.outerSphereMean *
		currentSphereDataPixel.pixelCount) - (2 *
		currentSphereDataPixel.outerSphereMean *
		currentSphereDataPixel.currentSum)
		)/(currentSphereDataPixel.pixelCount - 1);

		outputIt.Set( currentSphereDataPixel );

		++inputIt;
		++outputIt;
	}
}

template <class TInputImage, unsigned int VImageDimension >
void
	OuterSphereFilter<TInputImage,VImageDimension>::
	PrintSelf(std::ostream& os, Indent indent) const
{
	Superclass::PrintSelf(os,indent);
```

```
            os << indent << "Direction: " << m_Direction << std::endl;
      }

} // end namespace itk

#endif
```

itkInnerSphereFilter.h

```
#ifndef __itkInnerSphereFilter_h
#define __itkInnerSphereFilter_h

#include "itkImageToImageFilter.h"

// Shells and spheres specific includes
#include "itkShellIterator.h"
#include "itkSphereIterator.h"

#include <math.h>
#include <iostream>
#include <fstream>

#include <list>

#include <itkShapedNeighborhoodIterator.h>
#include "LinkedListPixel.h"

namespace itk
{

/** \class InnerSphereFilter
 * \Calculates the mean and standard deviation for the inner spheres,
comparing them with
 * the outer spheres to determine the direction to the most likely boundary,
as determined
 * by the z-value.
 *
 * This class is parameterized over the type of the input image and
 * the type of the output image.
 *
 * \ingroup
 */

template <unsigned int VImageDimension>
class ITK_EXPORT InnerSphereFilter : public ImageToImageFilter<
  itk::Image< SphereDataPixel<VImageDimension>, VImageDimension>,
  itk::Image< std::list<LinkedListPixel<VImageDimension>>, VImageDimension>
>
{
public:
  /** Standard class typedefs. */
  typedef InnerSphereFilter                          Self;
  typedef ImageToImageFilter<
        itk::Image< SphereDataPixel<VImageDimension>, VImageDimension>,
        itk::Image< std::list<LinkedListPixel<VImageDimension>>,
VImageDimension>>                                    Superclass;
  typedef SmartPointer<Self>                         Pointer;
  typedef SmartPointer<const Self>                   ConstPointer;

  /** Method for creation through the object factory. */
  itkNewMacro(Self);

  /** Run-time type information (and related methods). */
  itkTypeMacro(InnerSphereFilter, ImageToImageFilter);

  /** Some convenient typedefs. */
```

```
  typedef itk::Image< SphereDataPixel<VImageDimension>, VImageDimension>
InputImageType;
  typedef typename InputImageType::ConstPointer InputImagePointer;
  typedef typename InputImageType::RegionType   InputImageRegionType;
  typedef typename InputImageType::PixelType    InputImagePixelType;

  typedef itk::Image< std::list<LinkedListPixel<VImageDimension>>,
VImageDimension> OutputImageType;
  typedef typename OutputImageType::Pointer      OutputImagePointer;
  typedef typename OutputImageType::RegionType   OutputImageRegionType;
  typedef typename OutputImageType::PixelType    OutputImagePixelType;
  typedef typename OutputImageType::IndexType    OutputImageIndexType;

  std::list<LinkedListPixel<VImageDimension>> mylist;
  //list<LinkedListPixel<VImageDimension>> mylist;

  typedef itk::Image< std::list<LinkedListPixel<VImageDimension>>,
VImageDimension> LinkedListImageType;

   /** Set the direction in which to reflect the data. */
  itkGetConstMacro( Direction, unsigned int );
  itkSetMacro( Direction, unsigned int );
  itkSetMacro( MaxInnerSphereSize, unsigned int );
  itkSetMacro( MinInnerSphereSize, unsigned int );

  /** ImageDimension constants */
  itkStaticConstMacro(InputImageDimension, unsigned int,
                      InputImageType::ImageDimension);
  itkStaticConstMacro(OutputImageDimension, unsigned int,
                      OutputImageType::ImageDimension);

  //Shaped Neighborhood
  typedef ShapedNeighborhoodIterator< InputImageType > SNType;

#ifdef ITK_USE_CONCEPT_CHECKING
  /** Begin concept checking */
  /** End concept checking */
#endif

protected:
  InnerSphereFilter();
  virtual ~InnerSphereFilter() {};
  void PrintSelf(std::ostream& os, Indent indent) const;

  /** This method implements the actual reflection of the image.
   *
   * \sa ImageToImageFilter::ThreadedGenerateData(),
   *     ImageToImageFilter::GenerateData()  */
  void GenerateData(void);
  float ModifiedTTest( void );

private:
  InnerSphereFilter(const Self&); //purposely not implemented
  void operator=(const Self&); //purposely not implemented

  unsigned int m_Direction;
  int m_MaxInnerSphereSize;
```

```
  int m_MinInnerSphereSize;

};

} // end namespace itk

#ifndef ITK_MANUAL_INSTANTIATION
#include "itkInnerSphereFilter.txx"
#endif

#endif
```

itkInnerSphereFilter.txx

```
#ifndef __itkInnerSphereFilter_txx
#define __itkInnerSphereFilter_txx

#include "itkInnerSphereFilter.h"
#include "itkImageLinearIteratorWithIndex.h"
#include "itkImageLinearConstIteratorWithIndex.h"
#include "itkProgressReporter.h"

#include "itkListSample.h"

#include "itkCovarianceCalculator.h"
#include "itkVector.h"

#include "itkSymmetricEigenAnalysis.h"
#include "itkFixedArray.h"
#include "itkMatrix.h"

#define EigenAnalysis 1

namespace itk
{

	/**
	* Constructor
	*/
	template <unsigned int VImageDimension>
	InnerSphereFilter<VImageDimension>
		::InnerSphereFilter()
	{
		this->SetNumberOfRequiredInputs( 1 );
		m_Direction = 0;
	}

	/**

	* GenerateData goes through every pixel and calculates the mean at the
	pixel
	*/
	template <unsigned int VImageDimension>
	void InnerSphereFilter<VImageDimension>
		::GenerateData( void )
	{
		std::cout<<std::endl<<"Running InnerSphere Filter";
		int threshold = 100;//; -500
		//list<Cube> CubeData;

		// Input image pointer and output image pointer
		typename Superclass::InputImageConstPointer inputPtr = this-
		>GetInput();
		typename Superclass::OutputImagePointer outputPtr = this-
		>GetOutput(0);

		// Making the output the same size as the input
		outputPtr->SetRequestedRegion( inputPtr->GetRequestedRegion() );
		outputPtr->SetBufferedRegion( inputPtr->GetBufferedRegion() );
		outputPtr->SetLargestPossibleRegion( inputPtr-
		>GetLargestPossibleRegion() );
```

```
outputPtr->Allocate();

//Create a Image of Input Type to do manipulations
InputImageType::Pointer TempImage = InputImageType::New();
TempImage->SetRequestedRegion( inputPtr->GetRequestedRegion() );
TempImage->SetBufferedRegion( inputPtr->GetBufferedRegion() );
TempImage->SetLargestPossibleRegion( inputPtr-
>GetLargestPossibleRegion() );
TempImage->Allocate();

// Creating our iterators to go through the
output/input/tempImage
typedef ImageRegionConstIterator<InputImageType>
ConstInputIterator;
typedef ImageRegionIterator<OutputImageType> OutputIterator;
typedef ImageRegionIterator<InputImageType> TempImageIterator;


ConstInputIterator inputIt( this->GetInput(),  inputPtr-
>GetRequestedRegion() );
OutputIterator outputIt( outputPtr, outputPtr-
>GetRequestedRegion() );
TempImageIterator TempImIt( TempImage, TempImage-
>GetRequestedRegion() );

inputIt.GoToBegin();
TempImIt.GoToBegin();

// Giving the tempImage the information contained in the input
while(!TempImIt.IsAtEnd())
{
      TempImIt.Set(inputIt.Get());
      ++inputIt;
      ++TempImIt;
}

//Temporary SphereDataPixels to perform computations
SphereDataPixel<VImageDimension> currentSphereDataPixel;
SphereDataPixel<VImageDimension> tempSphereDataPixel;

//Set the iterators to the beginning of the images
inputIt.GoToBegin();
TempImIt.GoToBegin();
double maxoutermean = 0;
double minoutermean = 0;

//Create an image of linked list and allocate memory
LinkedListImageType::Pointer LinkedListImage =
LinkedListImageType::New();
LinkedListImage->SetRequestedRegion( inputPtr-
>GetRequestedRegion() );
LinkedListImage->SetBufferedRegion(inputPtr-
>GetBufferedRegion());
LinkedListImage->SetLargestPossibleRegion( inputPtr-
>GetLargestPossibleRegion() );
LinkedListImage->Allocate();
```

```
//Create an iterator for the linked list image
typedef ImageRegionIterator<LinkedListImageType>
LinkedListIterator;
LinkedListIterator LinkedListIt( LinkedListImage,
LinkedListImage->GetRequestedRegion() );

//Set the iterator to the beginning of the image
LinkedListIt.GoToBegin();

//Instantiate an object of LinkedListPixel class
LinkedListPixel<VImageDimension> CurrentLLPixel;
CurrentLLPixel.InitializeLinkedList();

std::list<LinkedListPixel<VImageDimension>> CurrentLinkedList;
std::list<LinkedListPixel<VImageDimension>> NeighbourLinkedList;

unsigned int element_radius = 1;
SNType::RadiusType radius;
radius.Fill(element_radius);

typedef ShapedNeighborhoodIterator< LinkedListImageType >
LLSNType;
LLSNType LLSNit(radius, LinkedListImage, LinkedListImage-
>GetRequestedRegion());

//Store pixel data for debugging
std::ofstream myfile ("TestingTorus156_10_27_A.txt");
if (myfile.is_open())
      myfile << "Index[0]\t[1]\t[2]\tScale\t
      InnerSphereMean\tOuterSphereMean\tInnerVariance
      \tOuterVariance\tZValue\tDirection[0]\t[1]\t[2]\n";

double bestInnerSphereMean;
double bestOuterSphereMean;
double bestInnerSphereVariance;
double bestOuterSphereVariance;
typedef itk::Vector <double, VImageDimension> DirectionType;
DirectionType tempDirection;

//Iterate over scale
for (unsigned int shellSize = (m_MinInnerSphereSize); shellSize
<= m_MaxInnerSphereSize; ++shellSize)
{
      std::cout<<std::endl<<"Scale = "<<shellSize;

      //Iterate over all pixels at each scale
      while(!TempImIt.IsAtEnd())
      {
            currentSphereDataPixel = TempImIt.Get();

            //Consider only those pixels whose intensity is
            //greater than the threshold set by the user

            if(currentSphereDataPixel.outerSphereMean >
            threshold)
            {
```

```
/**Indentation changed to accommodate the code in the document

// Temporary sums of the shell
double shellSum = 0;
double shellPixelCount = 0;
itk::ShellIterator <InputImageType> shellIt(const_cast<InputImageType*>(this-
>GetInput()), TempImIt.GetIndex(), shellSize, true);

itk::Offset<VImageDimension> myOffset;
//Compute data in a shell using shell iterator and accumulate the sum of
//pixel intensity
for(shellIt.GoToBegin(); !shellIt.IsAtEnd(); ++shellIt)
{
      if(shellIt.IsIndexWithinImage())
      {
            myOffset = shellIt.GetOffset();
            tempSphereDataPixel = shellIt.Get();

            //Compute statisttical data such as mean, variance
            currentSphereDataPixel.currentSum =
            currentSphereDataPixel.currentSum +
            tempSphereDataPixel.intensityValue;
            currentSphereDataPixel.currentSumSq =
            currentSphereDataPixel.currentSumSq
            +(tempSphereDataPixel.intensityValue *
            tempSphereDataPixel.intensityValue );

            ++currentSphereDataPixel.pixelCount;
      }
}

//Compute the mean and variance
// innerSphereMean = sphereSum/spherePixelCount;
currentSphereDataPixel.innerSphereMean =
currentSphereDataPixel.currentSum/currentSphereDataPixel.pixelCount;
                              currentSphereDataPixel.innerSphereVariance =
(currentSphereDataPixel.currentSumSq +
(currentSphereDataPixel.innerSphereMean *
currentSphereDataPixel.innerSphereMean * currentSphereDataPixel.pixelCount) -
(2 * currentSphereDataPixel.innerSphereMean *
currentSphereDataPixel.currentSum) )/(currentSphereDataPixel.pixelCount - 1);

itk::Offset<VImageDimension> shellBestPair;
itk::ShellIterator <InputImageType>
shellSpherePairIt(const_cast<InputImageType*>(this->GetInput()),
TempImIt.GetIndex(), (shellSize + m_MinInnerSphereSize), true);

//Setup sphere pairs to compute z value
for(shellSpherePairIt.GoToBegin(); !shellSpherePairIt.IsAtEnd();
++shellSpherePairIt)
{
      // Checks if the pixel is within the image, if not do nothing
      if(shellSpherePairIt.IsIndexWithinImage())
      {
            // Calculating the modified TTest
            tempSphereDataPixel = shellSpherePairIt.Get();
```

```
            currentSphereDataPixel.currentSpherePairTest = abs(
            (tempSphereDataPixel.outerSphereMean -
            currentSphereDataPixel.innerSphereMean)/
            sqrt((tempSphereDataPixel.outerSphereVariance) +
            (currentSphereDataPixel.innerSphereVariance)));

            //Find the optimum sphere pair. The one with the highest z-value
            if((currentSphereDataPixel.currentSpherePairTest >
            currentSphereDataPixel.bestSpherePairTest) &&
            (currentSphereDataPixel.innerSphereMean >
            tempSphereDataPixel.outerSphereMean))
            {
                  //Store data related to the optimum sphere pair
                  currentSphereDataPixel.bestSpherePairTest =
                  currentSphereDataPixel.currentSpherePairTest;
                  shellBestPair = shellSpherePairIt.GetOffset();
                  currentSphereDataPixel.bestSpherePair = shellBestPair;
                  currentSphereDataPixel.shellNumber = shellSize;
                  bestInnerSphereMean =
                  currentSphereDataPixel.innerSphereMean;
                  bestOuterSphereMean = tempSphereDataPixel.outerSphereMean;
                  bestInnerSphereVariance =
                  currentSphereDataPixel.innerSphereVariance;
                  bestOuterSphereVariance =
                  tempSphereDataPixel.outerSphereVariance;
            }
      }
}

// Getting the sums of the best sphere pairs...
double sumOfSqBestSpherePair = 0;
for (unsigned int counter = 0; counter < VImageDimension; counter++)
{
      sumOfSqBestSpherePair = sumOfSqBestSpherePair +
      (currentSphereDataPixel.bestSpherePair[counter] *
      currentSphereDataPixel.bestSpherePair[counter]);
}

// Creates our unit sphere pairs...
for (unsigned int counter = 0; counter < VImageDimension; counter++)
{
      currentSphereDataPixel.bestUnitSpherePair[counter] =
      currentSphereDataPixel.bestSpherePair[counter] / sqrt(
      sumOfSqBestSpherePair );
}
tempDirection = currentSphereDataPixel.bestUnitSpherePair;

//Store the data of optimum sphere pairs in a .txt file for debugging
myfile<<tempIndex2[0]<<"\t"<<tempIndex2[1]<<"\t"<<tempIndex2[2]<<"\t"<<shellS
ize<<"\t"<<bestInnerSphereMean<<"\t"<<bestOuterSphereMean<<"\t"<<bestInnerSph
ereVariance<<"\t"<<bestOuterSphereVariance<<"\t"<<currentSphereDataPixel.best
SpherePairTest<<"\t"<<tempDirection[0]<<"\t"<<tempDirection[1]<<"\t"<<tempDir
ection[2]<<"\t"<<"\n";

}
++TempImIt;
++inputIt;
```

```
}

//Create a itkShapedneighborhood of immediate neighbors to detect medial
//points

SNType::IndexListType ActiveIndexList;
SNType SNit(radius, TempImage, TempImage->GetRequestedRegion());
SNType::OffsetType Centeroffset = {(0,0,0)};
SNType::OffsetType offset1 = {(0,0,1)};
SNType::OffsetType offset2 = {(0,0,9)};
SNType::OffsetType offset3 = {(0,0,-3)};

SNit.ActivateOffset(Centeroffset);
SNit.ActivateOffset(offset1);
SNit.ActivateOffset(offset2);
SNit.ActivateOffset(offset3);

LLSNit.ActivateOffset(Centeroffset);
LLSNit.ActivateOffset(offset1);
LLSNit.ActivateOffset(offset2);
LLSNit.ActivateOffset(offset3);

LLSNit.GoToBegin();
LLSNType::Iterator LLInsideSNit;

//Initialize iterators to the beginning of the images obtained from the
//previous filter and also of the linked list image
SNit.GoToBegin();
LinkedListIt.GoToBegin();

//Store the data corresponding to the center of the neighborhood and the data
//of the neighboring pixels in different SphereDataPixel classes
SNType::Iterator InsideSNit;
SphereDataPixel<VImageDimension> SNPixel;
itk::Vector <double, VImageDimension> CenterPixelDirection;
itk::Vector <double, VImageDimension> NeighborDirection;
double DotProduct;
int centerShellSize;
double CenterPixelZValue;
int NeighborShellSize;
double NeighborPixelZValue;
SphereDataPixel<VImageDimension> SNNeighborPixel;

itk::Index<VImageDimension> tempIndex;

//Look for medial points by comparing the direction of the center of the
//neighborhood with the direction of its immediate neighbors
while( !SNit.IsAtEnd() )
{
      LLInsideSNit = LLSNit.Begin();
      LLInsideSNit++;
      InsideSNit = SNit.Begin();
      InsideSNit++; //To go to the center of the neighborhood
      SNPixel = InsideSNit.Get();

      CenterPixelDirection = SNPixel.bestUnitSpherePair;
      centerShellSize = SNPixel.shellNumber;
```

```
CenterPixelZValue = SNPixel.bestSpherePairTest;

//Go to the first neighbor of the center pixel as defined in the neighborhood

InsideSNit--;
LLInsideSNit--;

if(SNPixel.outerSphereMean >  threshold)
{
      //iterate through all the neighbors of the center of the neighborhood
      //as defined in the shaped neighborhood
      for (; InsideSNit != SNit.End(); InsideSNit++,LLInsideSNit++)
      {
            //Avoid considering the center of the neighborhood the second
            //time
            if (InsideSNit.GetNeighborhoodIndex() == 13)
            continue;

            SNNeighborPixel = InsideSNit.Get();

            //Entry into the linkedlist for each pixel is done here depending
            //on the direction of the neighboring pixels

            SNPixel = InsideSNit.Get();

            //Only consider those results whose scale matches the current
            //scale and compute the Dot Product
            if ((SNPixel.shellNumber == centerShellSize) &&
            (SNPixel.shellNumber == shellSize) )
            {
                  NeighborDirection = SNPixel.bestUnitSpherePair;
                  DotProduct = (CenterPixelDirection[0]*NeighborDirection[0]
                  + CenterPixelDirection[1]*NeighborDirection[1] +
                  CenterPixelDirection[2]*NeighborDirection[2]);

                  //If the dot product is negative and less than a threshold,
                  //mark both the center pixel and the corresponding neighbor
                  points as medial points at the current scale
                  if (DotProduct < -0.3 && SNPixel.bestSpherePairTest > 1.0
                  && CenterPixelZValue > 1.0)
                  {
                        CurrentLLPixel.Scale = SNPixel.shellNumber;
                        CurrentLLPixel.ZValue = SNPixel.bestSpherePairTest;
                        for (unsigned int counter = 0; counter <
                        VImageDimension; counter++)
                        {
                              CurrentLLPixel.Direction[counter] =
                              SNPixel.bestUnitSpherePair[counter];
                        }

                        CheckingLinkedList = LLInsideSNit.Get();
                        CheckingLLPixel.InitializeLinkedList();

                        //To store data at Neighbor Pixel
                        if(!CheckingLinkedList.empty())
                        CheckingLLPixel = CheckingLinkedList.back();
```

```
                        CurrentLinkedList = LLInsideSNit.Get();
                        CurrentLinkedList.push_back( CurrentLLPixel);

                        //Store only one result at each scale
                        if(!(CheckingLLPixel.Scale == SNPixel.shellNumber))
                        {
                              LLInsideSNit.Set(CurrentLinkedList);
                        }

                        //To store data at the Center Pixel
                        LLInsideSNit = LLSNit.Begin();
                        LLInsideSNit++;
                        CurrentLinkedList = LLInsideSNit.Get();
                        CheckingLinkedList = LLInsideSNit.Get();
                        CheckingLLPixel.InitializeLinkedList();

                        if(!CheckingLinkedList.empty())
                              CheckingLLPixel = CheckingLinkedList.back();

                        CurrentLLPixel.Scale = centerShellSize;
                        CurrentLLPixel.ZValue = CenterPixelZValue;
                        for (unsigned int counter = 0; counter <
                        VImageDimension; counter++)
                        {
                              CurrentLLPixel.Direction[counter] =
                              CenterPixelDirection[counter];
                        }
                        CurrentLinkedList.push_back( CurrentLLPixel);
                        if(!(CheckingLLPixel.Scale == SNPixel.shellNumber))
                        {
                              LLInsideSNit.Set(CurrentLinkedList);
                        }
                        break;

                  }
            }
      }
}
}

SNit++;
LinkedListIt++;
LLSNit++;

}

LinkedListIt.GoToBegin();
TempImIt.GoToBegin();
LLSNit.GoToBegin();

}

if(myfile.is_open())
      myfile.close();
```

```
#ifdef EigenAnalysis
//Eigen Analysis

//Create a list for storing eigen values
std::list<LinkedListPixel<MY_DIMENSION>> TempList;
std::list<LinkedListPixel<MY_DIMENSION>>::iterator TempListIt;
std::list<LinkedListPixel<MY_DIMENSION>>::iterator CurrentListIt;

LinkedListPixel<VImageDimension> tempLLPixel;
double tempScale;
double CurrentScale;

//define classes for covariance calculation and eigen value computation
typedef itk::Vector <double, VImageDimension> PixelDirectionType;
PixelDirectionType PixelDirection;
typedef itk::Statistics::ListSample< PixelDirectionType > SampleType;
SampleType::Pointer BankofPixels = SampleType::New();
BankofPixels->SetMeasurementVectorSize( VImageDimension );

typedef itk::Statistics::CovarianceCalculator< SampleType >
CovarianceAlgorithmType;
typedef CovarianceAlgorithmType::OutputType CovarianceMatrixType;
typedef itk::FixedArray< double, VImageDimension > EigenValuesArrayType;
typedef itk::Matrix< double,VImageDimension, VImageDimension >
EigenVectorMatrixType;
typedef itk::SymmetricEigenAnalysis< CovarianceMatrixType,
EigenValuesArrayType, EigenVectorMatrixType > SymmetricEigenAnalysisType;
EigenValuesArrayType EigenValues;
EigenVectorMatrixType EigenVectors;

//Create a .txt file to store the eigen values
std::ofstream EigenFile ("EigenAnalysisDataTorus156_10_27_A.txt");
if (EigenFile.is_open())
//define the heading for the file
EigenFile << "Index[0]\t[1]\t[2]\tScale\tEigenValue[0]\t[1]\t[2]\n";

itk::Index<VImageDimension> EigenTempIndex;

//Create a iterator to go through the linked list at each pixel
LLSNit.GoToBegin();
LLSNType::Iterator LLInsideSNit;

//Iterate through the image of linked lists
while(!LLSNit.IsAtEnd())
{
      LLInsideSNit = LLSNit.Begin();
      LLInsideSNit++;
      CurrentLinkedList  = LLInsideSNit.Get();
      EigenTempIndex = LLSNit.GetIndex();

      //Iterate through all the records in the linked list
      for(CurrentListIt = CurrentLinkedList.begin(); CurrentListIt !=
      CurrentLinkedList.end(); CurrentListIt++)
      {
            CurrentLLPixel  = *CurrentListIt;
            CurrentScale = CurrentLLPixel.getScale();
```

```
//Cluster all medial points within the inner sphere centered at
the current iterator index, whose radius is equal to the current
scale
for (unsigned int shellSize = (m_MinInnerSphereSize); shellSize
<= CurrentScale; ++shellSize)
{
      itk::ShellIterator <LinkedListImageType>
      shellIt(LinkedListImage, LLSNit.GetIndex(), shellSize,
      true);
      for(shellIt.GoToBegin(); !shellIt.IsAtEnd(); ++shellIt)
      {
            if(shellIt.IsIndexWithinImage())
            {
                  TempList = shellIt.Get();
                  for(TempListIt = TempList.begin(); TempListIt
                  != TempList.end(); TempListIt++)
                  {
                        tempLLPixel = *TempListIt;
                        tempScale = tempLLPixel.getScale();
                        if (tempScale == CurrentScale)
                        {
                              PixelDirection =
                              tempLLPixel.getDirection();
                              BankofPixels->PushBack(
                              PixelDirection );
                        }

                  }
            }
      }
}

//compute covariance matrix of the cluster created in the
//previous step using the CovarianceAlgorithm filter
CovarianceAlgorithmType::Pointer covarianceAlgorithm =
CovarianceAlgorithmType::New();
covarianceAlgorithm->SetInputSample( BankofPixels );
covarianceAlgorithm->Update();

//Compute the eigen valuesof the covariance matrix generated in
//the previous step
SymmetricEigenAnalysisType SymmetricEigenSystem(VImageDimension);
SymmetricEigenSystem.ComputeEigenValuesAndVectors
(*(covarianceAlgorithm->GetOutput()), EigenValues, EigenVectors
);

//Store the eigen values in a .txt file
EigenFile<<EigenTempIndex[0]<<"\t"<<EigenTempIndex[1]<<
"\t"<<EigenTempIndex[2]<<"\t"<<CurrentScale<<"\t"<<
EigenValues[0]<<"\t"<<EigenValues[1]<<"\t"<<EigenValues[2]<<"\n";

//Store the eigen values in the linked list
for (unsigned int counter = 0; counter < VImageDimension;
counter++)
{
      CurrentLLPixel.EigenValues[counter] = EigenValues[counter];
}
```

```
            *CurrentListIt = CurrentLLPixel;

      }
      LLSNit++;
}

//Close the eigen analysis .txt file
if(EigenFile.is_open())
      EigenFile.close();
#endif

outputIt.GoToBegin();
LLSNit.GoToBegin();

itk::Index<VImageDimension> tempIndex;
while(!outputIt.IsAtEnd())
{
      tempIndex = outputIt.GetIndex();
      if (tempIndex[0] == 18 && tempIndex[1] == 0 && tempIndex[2] == 19)
      LLInsideSNit = LLSNit.Begin();
      LLInsideSNit++;
      ++LLSNit;
      outputIt.Set(LLInsideSNit.Get());
      ++outputIt;
}

}

template <unsigned int VImageDimension>
float InnerSphereFilter< VImageDimension >::ModifiedTTest( void )
{

}

template <unsigned int VImageDimension>
void InnerSphereFilter<VImageDimension>
      ::PrintSelf(std::ostream& os, Indent indent) const
{
      Superclass::PrintSelf(os,indent);
}

}

#endif
```

BIBLIOGRAPHY

[1] A transformation for extracting new descriptors of shape H Blum, Models for the perception of speech and visual form, 1967

[2] Burbeck, C. A., Pizer, S. M., "Object representation by cores: Identifying and representing primitive spatial regions," Vision Research, vol. 35, no. 13, pp. 1917–1930, (1995)

[3] Pizer, S. M., Eberly, D. H., Morse, B. S., and Fritsch, D. S., "Zoom invariant vision of figural shape: The mathematics of cores," Comp. Vision Image Understanding, vol. 69, no. 1, pp. 55–71, (1998).

[4] G. Stetten, S. Pizer, Medial Node Models to Identify and Measure Objects in Real-Time 3D Echocardiography, IEEE Transactions on Medical Imaging, Vol. 18, No. 10, pp 1025-1034, Oct. 1999

[5] C.A. Cois, K. Rockot, J. Galeotti, R. Tamburo, G. Stetten, Shells and Spheres: A Framework for Variable Scale Statistical Image Analysis, CMU Robotics Tech Report#CMU-RI-TR-04-19, April, 19, 2006

[6] G. Stetten, S. Pizer, Automated Identification and Measurement of Objects via Populations of Medial Primitives, with Application to Real Time 3D Echocardiography,XVIth International Conference on Information Processing in Medical Imaging (IPMI), June 1999. Lecture Notes in Computer Science, vol. 1613, pp. 84-97.

[7] www.itk.org

[8] www.cmake.org