A Symmetry-Seeking Model for 3D Object Reconstruction

Using a Mesh of Particles

University of California
Los Angeles

Luis Ángel Larios C.
Miri Kim
Joyce Kuo

UNIVERSITY OF CALIFORNIA
Los Angeles

Computer Science
Visual Modeling

A SYMMETRY-SEEKING MODEL FOR 3D OBJECT RECONSTRUCTION
Using a Mesh of Particles

By:

Luis Ángel Larios Cárdenas
Miri Kim
Joyce Kuo

Professor:

Demetri Terzopoulos

Los Angeles, CA, March 21, 2012

A SYMMETRY-SEEKING MODEL FOR 3D OBJECT RECONSTRUCTION
Using a Mesh of Particles

Luis Ángel Larios Cárdenas, **Miri** Kim, and **Joyce** Kuo
University of California, Los Angeles

1. Introduction

We present an implementation model to reconstruct 3-dimensional objects with basic radial symmetry from 2-dimensional input images. The project began as an attempt to implement the approach introduced by Terzopoulos et al. in the paper "Symmetry-Seeking Models and 3D Object Reconstruction" [8]. Issues however, particularly with the fine balance of parameters related to curvature and mixed derivatives, led us to seek out a variation of the surface model. Our resulting approach is a static axis coupled with a particle mesh surface that deforms according to internal spring forces and external image and radial forces. This proposed model simplifies calculations and minimizes sensitivities of the original model while providing reasonably good results.

2. Related Work

The original symmetry-seeking model upon which our work is based [8], couples a deformable 2D parametric tube-like surface surrounding a deformable univariate spine. The boundaries of the tube surface deform according to forces provided by the image gradient, while the remainder of the surface deforms symmetrically by a radial force. Axial forces, associated with both the tube and spine, create a dynamic system between the two, while keeping the spine in an axial (centered) position. A contraction/expansion force allows further control of the deformation. The system is solved by balancing internal forces, characterized by a strain energy function, with external forces, characterized by the image, radial, axial, and contraction/expansion forces. We employ the fundamental strategy presented in this work.

Previous work on cloth simulation [2] provides the inspiration for our variation on the surface model. These simulation methods represent the simulated cloth as grid points connected by horizontal, vertical, and diagonal springs. Connection of the nodes in this manner allows deformation of the cloth while maintaining its structure without collapsing. We employ a basic spring mesh of grid points connected by horizontal and vertical springs. Integration of this basic mesh with the external forces described in the original symmetry-model above provides a reasonable deformable surface model suited to our purpose.

3. Model Formulation

Our model combines the ideas presented in the works outlined in the previous section. In particular, we make use of the model presented by [8], with the main modification being the replacement of the original tube surface with a spring mesh. In accordance, the internal strain energy function in the original system is replaced by spring forces to define the internal surface energy. In this section, we describe our model in detail.

3.1. A Deformable Particle Mesh

Our approach for a symmetry-seeking model consists of a mesh of *n* by *m* particles of unitary mass, interconnected by damped springs. Springs attach each particle to its four immediate neighbors [2] as observed in figure 1.

Homologous to a thin plate, a particle mesh[1] has an internal elastic, potential energy that is affected by the external forces modifying the relative particle positions. We represent this internal energy as spring forces, whose purpose is to restore the system to its natural state. The calculated spring force is proportional to the difference between the spring's rest length and the current distance of the interconnected particles. The following equations describe the damped-spring forces [5]:

$$F_{pq}^{spring} = k_{pq}^{spring}\left(l_{pq} - \left|\mathbf{x}_p - \mathbf{x}_q\right|\right)\frac{\mathbf{x}_p - \mathbf{x}_q}{\left|\mathbf{x}_p - \mathbf{x}_q\right|},$$

[1] We will use the terms *particle mesh* and *deformable surface* interchangeably since both serve the same purpose in the current implementation.

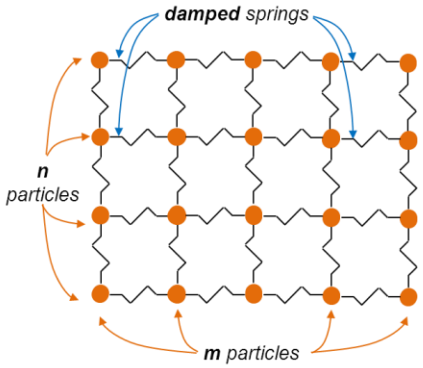

Figure 1. A mesh of particles connected by damped springs

$$F_{qp}^{spring} = -F_{pq}^{spring}$$

$$F_{pq}^{damping} = -k_{pq}^{damping}\left(\left(\dot{\mathbf{x}}_p - \dot{\mathbf{x}}_q\right)\cdot\frac{\mathbf{x}_p-\mathbf{x}_q}{|\mathbf{x}_p-\mathbf{x}_q|}\right)\frac{\mathbf{x}_p-\mathbf{x}_q}{|\mathbf{x}_p-\mathbf{x}_q|},$$

$$F_{qp}^{damping} = -F_{pq}^{damping}$$

where k_{pq}^{spring} and $k_{pq}^{damping}$ are the elastic and damping constants respectively, l_{pq} is the spring's rest length, $\mathbf{x}_p$ and $\mathbf{x}_q$ are the positions of the p^{th} and q^{th} particles connected by the spring, and $\dot{\mathbf{x}}_p$ and $\dot{\mathbf{x}}_q$ are the respective particle velocities.

The particles in the mesh move as a consequence of external forces acting upon them. Such motion ceases when those external forces are equal to the internal tension created by the spring forces attempting to return the springs to their rest length.

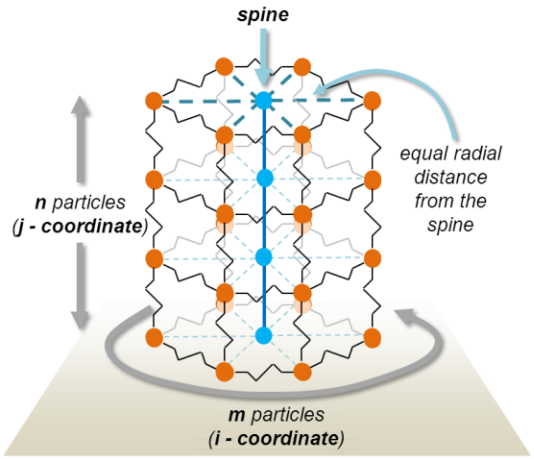

Figure 2. The deformable tube is formed by connecting edges of the particle mesh, surrounding the spine with radial symmetry

This condition of motionless particles is known as mechanical equilibrium. In our implementation, the deformable surface acquires the desired shape when mechanical equilibrium is reached.

Turning the Particle Mesh into a Deformable Tube

Departing from the original dynamic spine, our model establishes a static axis, $\mathbf{v}^s(j)$, that serves as support for the particle mesh. The axis is represented by a spline discretized into n points. Each of these points can be thought of as nodes in a univariate parametric coordinate system. The vector function $\mathbf{v}^s(j)$, then, represents a curve generator that yields 3D Euclidean coordinates for each of the points that make up the spine in the range $1 \leq j \leq n$, $j \epsilon \mathbb{N}$.

In the same manner as the spine, the particle mesh (figure 1) can be thought of as a bivariate parametric coordinate system, discretized into $n \times m$ nodes. Each point on this parametric surface $\mathbf{v}^t(i,j)$ represents 3D Euclidean spatial coordinates for the (i,j) particle, where $1 \leq i \leq m$ and $1 \leq j \leq n$, $i, j \epsilon \mathbb{N}$. A generalized cylinder is then constructed by surrounding the spine with the particle mesh. To create this deformable tube surface, we stitch together the ends of the i-coordinates of the particle mesh. The generalized cylinder is shown in figure 2.

This generalized cylinder can be thought of as a series of circular slices perpendicular to the spine, where nodes in the same slice share a common j parametric coordinate, while the i-coordinate varies from 1 to m (figure 3)[8]. That is, the j-coordinate describes the relative position along the spine, and the i-coordinate describes the position around the edge of a slice. Each node on a particular slice is set equidistant from the spine. The spine and the deformable surface are coupled via the parametric j-coordinate along the length of the tube, allowing nodes on a given slice to define a unique central point lying on the spine.

3.2. The Symmetry-Seeking Model

In addition to the internal elastic forces keeping the particle mesh together, each particle on the parametric surface is subject to external forces. These external forces include a radial-symmetry force, a contraction/expansion force, and a potential energy force generated by the image gradient.

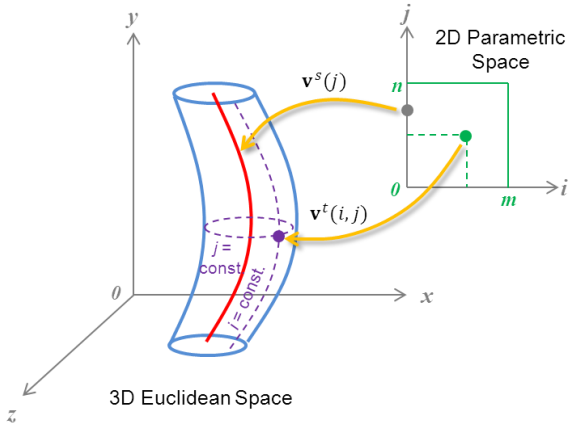

Figure 3. Parameterization of the spine and deformable surface

Before defining the external forces, it is useful to establish some functions that relate the fixed spine and the surrounding tube. Let $\mathbf{r}(i,j) = \mathbf{v}^T(i,j) - \mathbf{v}^s(j)$ be the *radial function* for each particle i on a circular slice located at the j^{th} level in the deformable cylinder. Then, we can set the *unit radial function* as $\hat{\mathbf{r}}(i,j) = \mathbf{r}(i,j)/\|\mathbf{r}(i,j)\|$, and the *mean radius* at the parametric coordinate *j=constant* as $\bar{r}(j) = \frac{1}{m}\sum_{i=1}^{m}\|\mathbf{r}(i,j)\|$.

The next equation describes the force that drives the tube towards radial symmetry around the spine. The constant $b(j)$ scales the strength of the force for all particles with the same parametric coordinate j (i.e. all particles on a circular slice):

$$\mathbf{f}_b^t(i,j) = b(j)(\bar{r}(j) - \|\mathbf{r}(i,j)\|)\hat{\mathbf{r}}(i,j)$$

Next, an external contraction/expansion force is used to further control the deformation, allowing shrinking of the tube in certain areas, while inducing expansion in other regions. This contraction/inflation force is calculated as follows:

$$\mathbf{f}_c^t(i,j) = c(j)\hat{\mathbf{r}}(i,j)$$

where $c(j)$ is a constant that defines an expansion force when $c(j) > 0$, and a contraction force when $c(j) < 0$.

Finally, given the image source $I(x,y)$, we extract a force from the image potential field, that tries to match the projection of the 3D deformable tube to the boundaries of the 2D object. We calculate the image potential field for the surface according to the following expression:

$$P[\mathbf{v}^T(i,j)] = \varphi(i,j)\|\nabla\{G_\sigma * I(\Pi[\mathbf{v}^T(i,j)])\}\|$$

where G_σ is a Gaussian kernel of characteristic width σ, whose purpose is to extend the capture range of the 2D object's edges. The Gaussian kernel is used to convolve the image at locations corresponding to the orthographic projection of the surface, $\Pi[\mathbf{v}^T(i,j)]$. The potential field is then obtained by computing the magnitude of the gradient of the convolved image. The weighting term $\varphi(i,j)$ allows the image potential field to affect only those particles whose projection lies near the boundaries of the surface's projection, and is calculated:

$$\varphi(i,j) = \begin{cases} 1 - |\mathbf{k}\cdot\mathbf{n}(i,j)|, & if\ |\mathbf{k}\cdot\mathbf{n}(i,j)| < 0.05 \\ 0, & otherwise \end{cases}$$

where $\mathbf{k}$ is the normal vector to the image plane, and $\mathbf{n}(i,j)$ is the normal to the surface at the location of the (i, j) particle. This last expression assigns an image potential only to particles with normal vectors that are closely parallel to the image plane. This restriction indicates that the image force is only exerted on particles along the silhouette of the deformable tube.

Having all of the external forces defined, the equation that describes the motion of the symmetry-seeking model is given by:

$$\gamma\frac{\partial\mathbf{v}^t}{\partial t} = \mathbf{f}_{sp}^t + \frac{\partial P}{\partial\mathbf{v}^t} + \mathbf{f}_b^t + \mathbf{f}_c^t$$

where $\mathbf{f}_{sp}^t(i,j) = \sum_{q\neq p}\left(F_{pq}^{spring} + F_{pq}^{damping}\right)$ is the set of internal forces affecting the $p = (i, j)$ particle on the deformable surface, and γ is the friction or dissipation factor. The goal of the symmetry-seeking model is to solve this equation of motion to equilibrium, as the forces drive the deformable surface towards the desired shape.

4. Implementation and Results

To solve the equation of motion describing the deformable surface, we implement an explicit Euler integration step in an iterative fashion for each particle. The complete process is described by algorithm 1, with details regarding each step given below.

// Initialization
Read the image source containing the object to reconstruct
Create the spine from user-clicked control points on the image plane
Initialize a generalized cylinder $\mathbf{v}(i,j)$ surrounding the spine
Set the initial velocity $\dot{\mathbf{v}}(i,j) = \mathbf{0}$
Set the average elastic energy of the surface $E = 0$
Initialize the b constant for the $\mathbf{f}_b^t$ force
Set the initial values for time step size Δt, friction γ, and Gaussian characteristic width σ

// Multi-resolution loop
While $\sigma > 1$ **do**
    Blur the gray-scale image source with a Gaussian of characteristic width σ
    Compute the image potential field for the blurred image
    Calculate the c constant for the $\mathbf{f}_c^t$ force depending on current σ value
    Calculate the spring and damping force constants, k_{pq}^{spring} and $k_{pq}^{damping}$, depending on σ
    Compute the average $speed$ of the surface // how much the surface is deforming

    // Update loop
    While $speed > \varepsilon$ and $iterations < iterations_{threshold}$ **do**
        If $E > E_{threshold}$ and $\sigma < \sigma_{threshold}$ **then**
            Relax springs in the deformable surface //update rest length
        End **if**

        Compute $\mathbf{f}_{sp}^t$ and the new E
        Compute $\mathbf{f}_b^t$ and $\mathbf{f}_c^t$ and scale them with their respective constants
        Compute the force $\frac{\partial P}{\partial \mathbf{v}}$ from the image potential field

        Set $\mathbf{v}_{old} = \mathbf{v}$
        Update the velocity $\dot{\mathbf{v}} = \dot{\mathbf{v}} + \frac{\Delta t}{\gamma}\mathbf{f}_{sp}^t + \mathbf{f}_b^t + \mathbf{f}_c^t + \frac{\partial P}{\partial \mathbf{v}}$
        Update the position of the particles $\mathbf{v} = \mathbf{v}_{old} + \gamma\dot{\mathbf{v}}$

        Compute the average $speed$ of the surface
    End **while**
    Decrease the value of σ
End **while**

Algorithm 1. Solving the system to equilibrium

4.1. Initializing the System

Once an input image has been read, a user provides a set of 5 control points to initialize the spine. The control points define 4 segments or cubic splines to be interpolated. Based on these control points, the system generates 25 points representing discrete values of the parametric curve $\mathbf{v}^s(j)$. Figure 4 shows the initial spine drawn on a grayscale image containing the object we wish to reconstruct.

Each of the 25 points along the spine defines the center of 25 circles stacked upon one another. Every circle contains 44 connected particles, representing the discretized points of the parametric 2D surface $\mathbf{v}^t(i,j)$ for a constant j-coordinate value. These particles are further connected with their respective *j+1* and *j-1* level i^{th} neighbors, building the deformable surface. By connecting particles $i = 1$ and $i = m$ at each j of the parametric surface, a deformable tube is created surrounding the spine, as it is shown in figure 5.

During the initialization step, we set the velocity and speed of the deformable tube to zero, representing equilibrium in the starting state. At this stage, the cumulative potential energy of the springs interconnecting the particles is also set to zero. Thereafter, it is recalculated using the following equation:

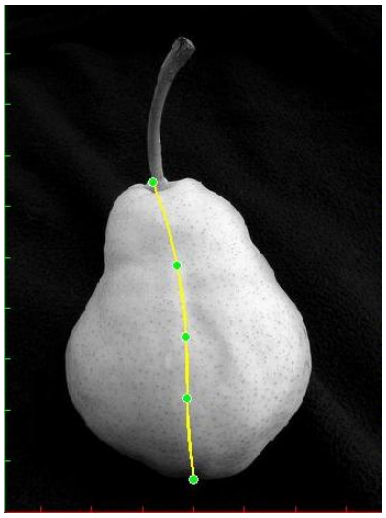

Figure 4. Initial spine generated by user-clicked control points

$$E_{pq} = \frac{1}{2} k_{pq}^{spring} \left(l_{pq} - |\mathbf{x}_p - \mathbf{x}_q| \right)^2$$

where p and q are neighboring particles attached by a spring whose rest length is l_{pq}.

Next, the b constant is assigned a value for scaling the radial force $\mathbf{f}_b^t$. Currently, b must be in the range (0, 0.4] in order to avoid counteracting the image force that deforms the surface into the desired shape.

At the end of the initialization step, we set the values for the time step, friction, and the characteristic width (or standard deviation) of the Gaussian kernel (used later). The time step must be chosen carefully in order to prevent the system from becoming unstable. The current time step is set to 0.001, which allows the simulation to steadily converge in under 15 minutes. The implementation of the symmetry-seeking model is then divided into two main loops that compel the surface to acquire the target shape: a multi-resolution loop and an update loop.

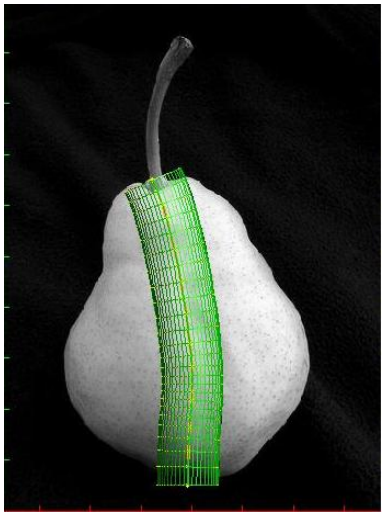

Figure 5. The initial deformable tube

4.2. The Multi-Resolution Loop

The multi-resolution loop embeds the update loop and has the purpose of ensuring that the forces generated by the image gradient are able to affect the surface. The initial σ depends on characteristics of the image, and can be as great as 150 pixels for shapes that vary in width (e.g. the pear in figure 4). The multi-resolution loop begins with a large σ in order to increase the capture range of the 2D object's edges, then decreases by successive divisions when the embedded update loop completes. As σ becomes smaller, the edges of the object in the image source become finer, allowing the 3D model to reconstruct detailed contour information.

Inside the multi-resolution loop, we first convolve the gray-scale image source with a Gaussian kernel of width σ. Next, the blurred image is used to compute a potential field, defined by the magnitude of the gradient of the image. To compute the gradient of discrete pixels, we implement a convolution with the Sobel operator [3], thus obtaining two images representing an approximation to the derivatives in the horizontal (*Ix*) and vertical (*Iy*) directions. Given both derivatives, we compute their magnitude and combine them into one image corresponding to the potential field for the current σ value. Figure 6 illustrates the potential field for the pear's image after being blurred with a Gaussian kernel whose $\sigma = 150$.

Having computed the image potential field, we again convolve this image-potential with the Sobel operator. As a result, we obtain, once more, two component images, *Ix* and *Iy*, which now represent the external forces for driving the particle mesh towards its desired shape.

The multi-resolution loop also offers the advantage of adjusting the c and spring/damping constants depending on the stage of reconstruction. If $\mathbf{f}_c^t$ is an *expansion* force, we want this force to have less effect on the deformable tube as σ gets smaller, so that the surface will not surpass the target boundaries. We achieve this by linearly interpolating the c constant between some maximum value (approximately in the range of [0.001, 0.003]) and zero, with respect to the current σ value. When $\mathbf{f}_c^t$ is

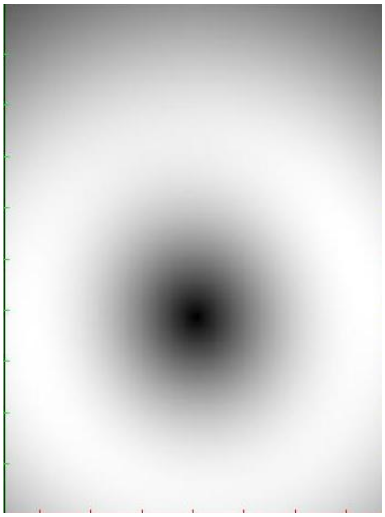

Figure 6. Image potential field resulting from computing the magnitude of the gradient of the image source convolved with a Gaussian with $\sigma = 150$

a *contraction* force, we keep the force fixed by setting c to a constant value in the range [-0.04, -0.01]. We also note that since the c constant is directly related to the shape to be reconstructed, its maximum value must be chosen carefully to prevent the system from collapsing in areas where the expansion/contraction force strongly conflicts with the internal restoration forces.

The spring and damping constants need to be small for a wide σ, and large when the deforming tube nears the edges of the object image. We found it to be unnecessary to interpolate the spring and damping forces as in the case of the expansion c constant. Here, we simply shift the values from small to large when σ becomes less than some threshold, so that springs reduce their tendency to oscillate as they attempt to restore their natural rest lengths. We have found that initial values in the range [1, 4], and ending values in the range (4, 10] in the last stage of reconstruction work well. Spring and damping constants are also highly dependent on the image source. When these constants are correctly tuned, we should see the tube surface move towards the desired boundaries, and remain there once reached.

Finally, before entering the update loop, we calculate the speed of deformation as our termination criteria. The surface speed is computed as the average of all of the particles' velocity magnitude. We define the system to be in *equilibrium* when we reach a mean speed of 0.001 or less.

4.3. The Update Loop

The update loop is an iterative process in which the positions of the particles of the deformable surface are updated according to the effect of the internal and external forces. This loop executes until the particle mesh reaches equilibrium, or until a maximum number of iterations is reached. We set this maximum in order to avoid running the simulation indefinitely. Depending on the shape to reconstruct, we set the maximum number of iterations anywhere from 100 to 300. In practice, the maximum number of iterations is normally reached before converging to equilibrium.

The first step of the update loop is to relax the springs. We relax the springs by resetting their natural rest length to the average of their current length and rest length, thereby liberating part of the accumulated internal potential energy and reducing the tension created from the deformation. In this manner, particles are able to continue towards their desired position without being restricted by the internal tension. In order to keep the internal and external forces in balance, we relax the springs only when the average potential energy reaches a maximum threshold. Relaxing the springs must take place during the first stages of reconstruction (when σ is large), to prevent the surface from developing undesirable wrinkles and protrusions, indicating an over-relaxation that detrimentally affects the quality of the final 3D model. Both the maximum potential energy and σ thresholds must be chosen carefully according to the specific image in order to prevent the system from collapsing (lack of relaxation) or going beyond the target boundaries (over-relaxation).

Whether the springs are relaxed or not, the next step is to compute the internal spring forces $\mathbf{f}_{sp}^{t}$ and the accumulated elastic energy E, given the current positions of the particles. These forces represent the amount of tension in the deformable tube, and must be large enough to counterbalance the external forces.

The external forces $\mathbf{f}_{b}^{t}$ and $\mathbf{f}_{c}^{t}$ are then calculated to keep the particles radially symmetric with respect to the spine, and to inflate/contract the tube in specific regions. Both $\mathbf{f}_{b}^{t}$ and $\mathbf{f}_{c}^{t}$ are multiplied by their respective constants in order to potentiate these external forces in driving the model towards its final shape.

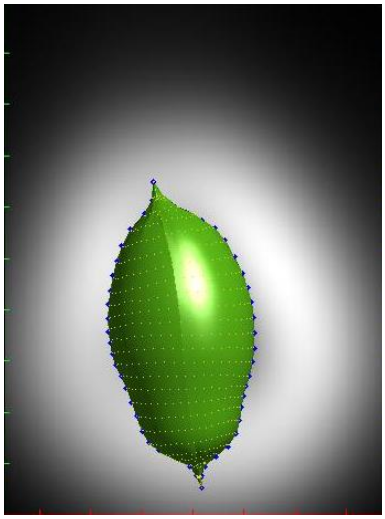

Figure 7. Image forces acting on particles along the surface contour

The previously obtained image potential field and its gradient (*Ix* and *Iy*) now provide the driving image force $\frac{\partial P}{\partial \mathbf{v}}$. The image force is calculated such that it only affects particles whose normal vector at the tube surface [7] is essentially perpendicular to the image plane normal (the unit vector parallel to the *z* axis in our implementation). That is, the image force is only applied to particles whose projection is along the contour of the 3D deformable tube. The image force attempts to pull these particles towards regions in the potential field that have large values (brighter areas in figure 6). Since both *Ix* and *Iy* are discretized spaces (i.e. made of pixels), the force assigned to the particles' continuous Euclidean positions are computed using bilinear interpolation. However, these interpolated forces are not applied directly to the particles as the particles may lose their relative position along the slice (parametric *j* coordinate). Instead, the image forces are projected onto the unit radii that emanate from the spine towards the particles, so that the points in space will move mostly parallel to a disc corresponding to their j^{th} level in the generalized cylinder. For example, movement will be mostly horizontal for an object with a vertical axis. Figure 7 illustrates the particles along the surface silhouette as blue dots, indicating that they are being pulled towards the bright areas due to image forces acting on them.

Finally, after computing all of the internal and external forces, the particles' new velocities are updated using an explicit Euler integration step, solving the equation of motion given in section 3.2. The particles' positions are then updated according to the new velocities (clamped to 1 pixel-norm to avoid disproportional motion among particles). The current deformation speed is again computed to test the termination condition. Figure 8 illustrates various stages during the reconstruction process of a pear. The snapshots are taken during the first iterations of the update loop, after initialization of constants during the embedding multi-resolution loop.

4.4. Results

The symmetry-seeking model is implemented in MATLAB [4], which provides tools for image processing, matrix computation, user interaction, and visualization [6]. Using this platform, we were able to reconstruct a pear, a banana, and even a mango that had portions occluded by another object. At the end of the modeling process, we textured map the 3D models for a more realistic appearance as shown in figures 9, 10, and 11.

As it is observed in figure 10, our symmetry-seeking model reconstructs objects lying horizontally or

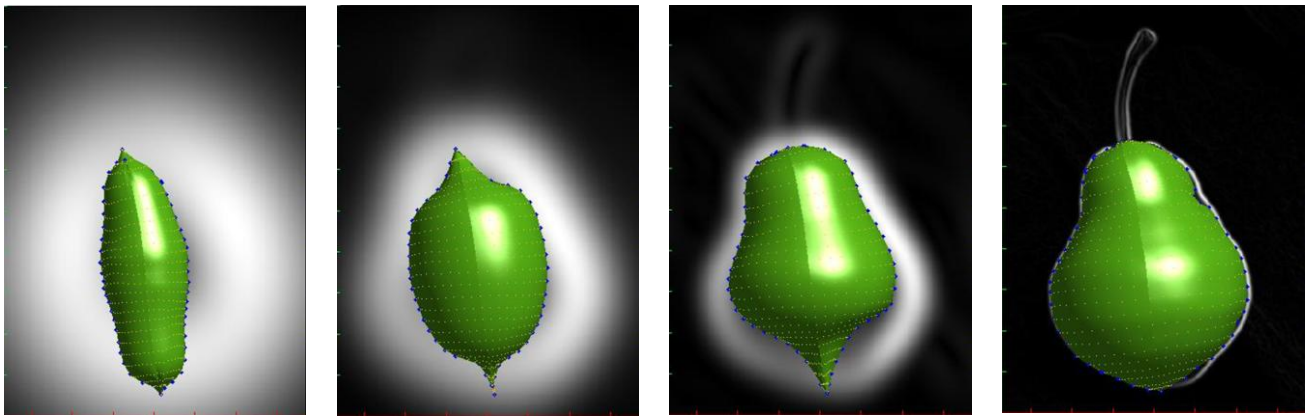

Figure 8. Evolution of the reconstruction process of a pear

Figure 9. Textured, reconstructed pear

vertically without distinction. Furthermore, the radial-symmetry force $\mathbf{f}_b^t$ allows us to generate 3D models of objects whose silhouettes may be partially occluded in the original image (figure 11). This force assumes that these objects are symmetric around the spine, and has the ability to generate the 3D model so long as at least one half of the object boundary, running laterally to the spine, is visible for computing the image potential field.

We have included the MATLAB code in the attached appendix (with parameters tuned for generating the banana in figure 10), as well as three accompanying videos illustrating the reconstruction process of the three objects. The videos show how the tube surface evolves over time, until it is textured for giving it an improved level of realism.

5. Conclusions and Future Work

In contrast to the original symmetry-seeking model [8], our implementation contains a stationary axis. As a result, the accuracy of the reconstructed object is highly dependent on the initialization of the spine. This dependency requires special attention from the user when specifying the spine's base points. However, removing the dynamics of the spine allows us to simplify the reconstruction process and obtain a higher consistency of results for the simple objects we want to reconstruct. The elimination of axial forces for both the spine and the tube surface minimizes the sensitive parameter tuning required to find the delicate balance between symmetry, inflation/contraction, and spring forces constants. By trimming the number of parameters to adjust, we exchange a harder-to-find accuracy for simplicity and stability.

The second and main simplification in our approach is a replacement of the thin plate deformable surface [8], with a mesh of particles. Employing this mesh speeds up the reconstruction process by avoiding the complex computations of curvature and mixed derivatives, as well as the associated parameter sensitivities. These second order derivatives were primarily necessary to calculate the thin plate's stiffness matrix at every time step, determining elasticity and internal bending forces in such a surface. In that approach, issues arise primarily due to unequally spaced nodes in the surface. These non-uniformly distributed nodes distort the meaning of the first- and second-order derivatives approximated by finite differences. The use of the particle mesh thus allows us to circumvent issues of relocating or resampling nodes in order to effectively approximate

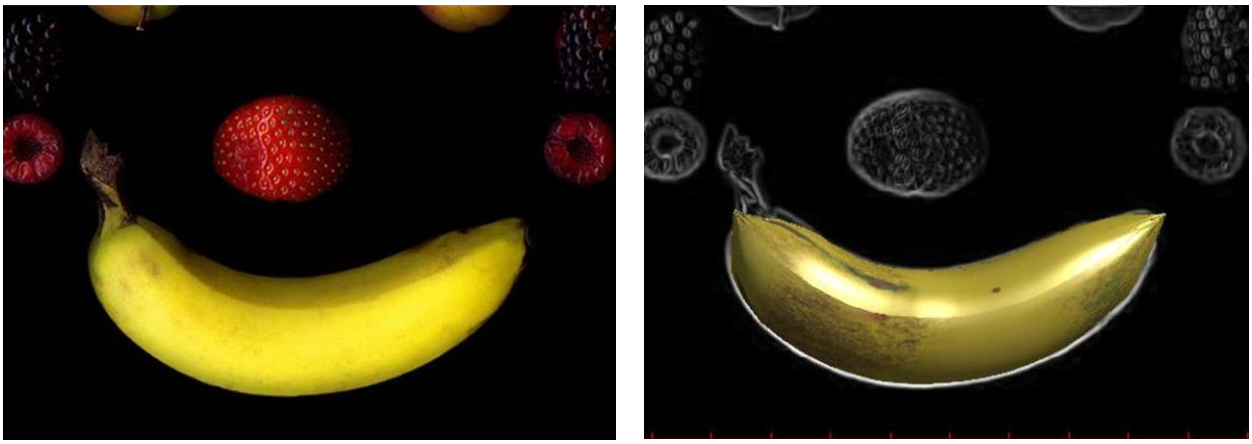

Figure 10. A textured, reconstructed banana (right) from its image source (left)

Figure 11. A textured, reconstructed mango (right) from its image source (left)

the required derivatives. We are also able to eliminate the additional time complexity involved in computing the stiffness matrix at every iteration. In our approach, tension previously defined by the stiffness matrix now comes from the elastic potential energy calculated from the springs making up the particle mesh. Ultimately, our approach is essentially equivalent to a thin plate with curvature set to zero.

As shown in the example figures and accompanying videos, our implementation successfully reconstructs 3D models from 2D input images. The main limitation in our implementation is the necessity to specifically tune parameters according to the input image. However, when set properly, our symmetry-seeking implementation is able to generate 3D models that capture detailed profile information from the image source. When enough information is available, our model is even capable of reconstructing partially occluded objects.

Finally, we mention some possible extensions to the current implementation. Firstly, a dynamic spine would increase the accuracy level of reconstructed objects. This could be accomplished similarly to the strategy employed in the original model [8] with coupled axial forces. Secondly, an automated spine initialization process may yield more consistent results that are less reliant on user input. Additional image segmentation techniques could also be integrated to further increase accuracy. These improvements would provide a robust model, less sensitive to input images, and allow a wider range of possible reconstruction shapes.

6. References

[1] M. Kass, A. Witkin, and D. Terzopoulos. (1987). *Snakes: Active Contour Models*. International Journal of Computer Vision 1(4).

[2] J. Lander. (1999). *Devil in the Blue Faceted Dress: Real-Time Cloth Simulation*. Game Developer Magazine, May Edition.

[3] Y. Ma, S. Soatto, J. Kosecka, and S. Sastry. (2004). *An Invitation to 3-D Vision: From Images to Geometric Models*. Springer-Verlag.

[4] The MathWorks Inc. (1994 - 2012). *MATLAB Software, Product Documentation, and Tutorials*. http://www.mathworks.com/products/matlab/

[5] R. Parent. (2008). *Computer Animation: Algorithms and Techniques*. Second Edition. Morgan Kaurfmann.

[6] D. S. Parker. (2000). *Exploring the Matrix: Adventures in Modeling with MATLAB*. Academic Publishing Services, UCLA.

[7] E. Swokowski. (1989). *Cálculo con Geometría Analítica*. Spanish Second Edition. Grupo Editorial Iberoamérica.

[8] D. Terzopoulos, A. Witkin, and M. Kass. (1987). *Symmetry-Seeking Models and 3D Object Reconstruction*. International Journal of Computer Vision 1(3).

A SYMMETRY-SEEKING MODEL FOR 3D OBJECT RECONSTRUCTION

Using a Mesh of Particles

Appendix

The following code corresponds to MATLAB functions where the constant values for the forces have been tuned to reconstruct a banana (shown in figure 10).

Function to Create the GUI

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Visual Modeling Project.
% Main file to construct GUI.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function launchApplication2(fileName)
    %%%%%%%%%%%%%%%%%%%%%%% Load image to work with %%%%%%%%%%%%%%%%%%%%%%%%%
    grayImage = loadImage(fileName);
    [height width] = size(grayImage);           %Get size to create the GUI.

    %%%%%%%%%%%%%%%%%%%%%%%%% Setting up the GUI %%%%%%%%%%%%%%%%%%%%%%%%%%%%
    figureHandler = figure(1);                  %Window workspace.
    set(figureHandler, 'Toolbar', 'figure',...  %Defines properties for the
                                                %figure.
        'OuterPosition', [50 50 width+300 height+150],...
        'Resize', 'off');

    %%%%%%%%%%%%%%%%%%%%%%%%%% Axes for drawing %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    imgAxes = gca;                              %Axes for image display.
    set(imgAxes, 'DataAspectRatio', [1 1 1],...
        'XColor', 'y',...
        'YColor', 'y',...
        'XAxisLocation','top',...
        'YAxisLocation','right',...
        'XLim', [0 width],...
        'YLim', [0 height],...
        'YDir', 'reverse',...
        'HandleVisibility', 'on',...
        'NextPlot', 'replacechildren');
    colormap(imgAxes, gray(256));
    imagesc(grayImage, 'Parent', imgAxes);      %Display initial image.

    surfaceAxes = axes('Position',get(imgAxes,'Position'),...
        'DataAspectRatio', [1 1 1],...          %Create axes for surface
                                                %display.
        'Color','none',...
        'XColor','r',...
        'YColor','g',...
        'ZColor','b',...
        'XLim', [-width/2 width/2],...
        'YLim', [-height/2 height/2],...
        'ZLim', [-width/2 width/2],...
        'HandleVisibility', 'on',...
        'NextPlot', 'replacechildren');
```

```
    %%%%%%%%%%%%%%%%%%%% Adding buttons for interaction %%%%%%%%%%%%%%%%%%%%
    uicontrol(figureHandler, 'Style', 'pushbutton',...
        'String', 'Provide Points',...
        'Position', [20 200 100 30],...
        'Callback', {@clearButtonCallback, surfaceAxes, imgAxes, grayImage});
                                                %Callback with parameter.

    uicontrol(figureHandler, 'Style', 'pushbutton',...
        'String', 'Start Simulation',...
        'Position', [20 150 100 30],...
        'Callback', {@startButtonCallback, surfaceAxes, imgAxes, grayImage});
                                                %Callback with parameter.
end

%%%%%%%%%%%%%%%%%%%%%%% GUI components callbacks %%%%%%%%%%%%%%%%%%%%%%%%%

% Callback function for the clearButton button.
function clearButtonCallback(hObj, event, surfaceAxes, imgAxes, img)
    cla(surfaceAxes);                   %Clean plotting area in both axes.
    cla(imgAxes);

    %Redraw image passed as parameter.
    imagesc(img, 'Parent', imgAxes);

    %Declare static variables to use by reference in other functions.
    global X Y Z;           %Snake point coordinates.
    P = click_image(5);     %Get 5 points from clicking on the current axes.
    [X Y] = init_spine(P, 25);
    Z = zeros(size(X,1), 1);

    %Draw initial snake.
    plot(X, Y, 'LineWidth', 2, 'Color', 'y', 'Parent', surfaceAxes);
    set(surfaceAxes, 'NextPlot', 'add');                %Hold on.
    plot(P(1,:), P(2,:), 'Line', 'none', 'Marker', 'o', 'MarkerEdgeColor', 'w',
'MarkerFaceColor', 'g');
    set(surfaceAxes, 'NextPlot', 'replaceChildren');    %Hold off.

    %Generate global matrices with the size of the surface.
    computeGlobalMatrices(size(X,1), size(X,1)+19);   %First rows and then
                                                      %columns.
end

% Callback function for the simulation button.
function startButtonCallback(hObj, event, surfaceAxes, imgAxes, img)
    cla(surfaceAxes);                   %Clean plotting area in both axes.
    cla(imgAxes);

    %Call the function for the simulation.
    symmetry2(surfaceAxes, imgAxes, img);
end
```

Function to Load an Image

```
function grayImage = loadImage(fileName)
%Function to load a color RGB image and return it in a gray-scale
%equivalance.
I = imread(fileName);
```

```
grayImage = colorToGrayScale(I);
grayImage = grayImage(:, :, 1);     %Return just one channel.
```

Function to Transform an RGB Image into Gray-Scale

```
function GrayA = colorToGrayScale(A)
% colorToGrayScale(A) produces the gray-scale image version of the color 3D
% matrix A.
[R G B] = imageToRGB(A);
Maximum = max(R, max(G, B));
GrayA = rgbToImage(Maximum, Maximum, Maximum);
```

Function to Decompose a 3D-Matrix Image into its RGB Channels

```
function [R G B] = imageToRGB(A)
% Function that decompose an image 3D-matrix A into its RGB channels in 3
% 2D-matrices R G B of double values. A can have double or uint8 values.
R = double(A(:, :, 1)); %Extracts red.
G = double(A(:, :, 2)); %Extracts green.
B = double(A(:, :, 3)); %Extracts blue.
if(isinteger(A))  %Behavior of function depends on type received.
    R = R/255.0;
    G = G/255.0;
    B = B/255.0;
end;
```

Function to Combine RGB Channels into A 3D-Matrix Image

```
function A = rgbToImage(R, G, B)
% rgbToImage(R,G,B) compose an image 3D-matrix 'A' from its RGB channels
% given by double-value 2D-matrices R G B. Resulting 'A' contains uint8
% values in [0,255].

R = min(1.0, R);    R = max(0.0, R);    %Verifies values inside [0,1].
G = min(1.0, G);    G = max(0.0, G);
B = min(1.0, B);    B = max(0.0, B);

A(:, :, 1) = uint8(255*R); %Inserts red.
A(:, :, 2) = uint8(255*G); %Inserts green.
A(:, :, 3) = uint8(255*B); %Inserts blue.
```

Function to Get Points from Clicking on the Image Plane

```
function [x] = click_image(npts)
% Function that returns the coordinates of the points where the user
% clicks.
if(nargin==0)
n_points = input('How many points?');
[x,y] = ginput(n_points);
elseif(nargin==1)
n_points = npts;
[x,y] = ginput(n_points);
end;

x=[x,y]';
```

Function to Initialize the Spine

```
function [ fx, fy ] = init_spine( user_pts, n_spine_pts )
%Initializes a spine (as a cubic spline) given a 2xn matrix of n user-clicked
% points
%   @user_pts: 2xn matrix of n user-clicked points
%   @n_spine_pts: total number of discretized points along the spine.
%   @fx: a column vector of x-coordinates (of size n_spine_pts)
%   @fy: a column vector of y-coordinates (of size n_spine_pts)

%Extract input arguments
n_usr_pts = length(user_pts);
x_in = user_pts(1,:);
y_in = user_pts(2,:);

%Parameterize and create the spine
h = (n_usr_pts-1)/(n_spine_pts-1);      %"Parametric distance" between points
                                        %(NOT normalized from 0 to 1)
s = (0:n_usr_pts-1)';                   %Used to parameterize user/control
                                        %points
t = (0:h:n_usr_pts-1)';                 %Parametric argument
fx = spline(s,x_in,t);                  %Get x-coordinates
fy = spline(s,y_in,t);                  %Get y-coordinates
```

Function to Compute the Global Matrices

```
function computeGlobalMatrices(ni, mj)
%Function to compute the global matrices.
%'ni' is the number of rows. 'mj' is the number of columns of the grid.
global n m;             %Global dimensions.
global Actrln Actrlm;   %Matrices for first derivatives.
global ImgPlaneNormal;  %3D matrix that contains the normal vector to the
                        %image plane.

%%%%%%%%%%%%%%%%%% Initialize number of rows and columns %%%%%%%%%%%%%%%%%%
n = ni;
m = mj;

%%%%%%%%%%%%%%%%%%%%%% Matrices for first derivatives %%%%%%%%%%%%%%%%%%%%%%
%Central differences.
Actrln = diag(ones(n-1, 1), 1);   %Upper diagonal.
Actrln = Actrln + diag(ones(n-1, 1), -1)*-1;   %Lower diagonal.
Actrln(n,1) = 1;
Actrln(1,n) = -1;
Actrln = 0.5 * sparse(Actrln);

Actrlm = diag(ones(m-1, 1), 1);   %Upper diagonal.
Actrlm = Actrlm + diag(ones(m-1, 1), -1)*-1;   %Lower diagonal.
Actrlm(m,1) = 1;
Actrlm(1,m) = -1;
Actrlm = 0.5 * sparse(Actrlm);

%%%%%%%%%%%%%%%% 3D matrix with normals to the image plane %%%%%%%%%%%%%%%%
ImgPlaneNormal = zeros(n, m, 3);
ImgPlaneNormal(:,:,1) = sparse(zeros(n, m));
ImgPlaneNormal(:,:,2) = sparse(zeros(n, m));
ImgPlaneNormal(:,:,3) = ones(n, m);
```

Function for the Symmetry-Seeking Model

```
function symmetry2(surfaceAxes, imgAxes, img)
% Function to simulate the symmetry seeking model.

%Height and width of the underlying image for frame transformation.
[height width] = size(img);

%%%%%%%%%%%%%%%%%%%%%%%% Creating the 3D surface %%%%%%%%%%%%%%%%%%%%%%%%
global X Y Z n m;           %Access to spine coordinates and size variables.

[Xij Yij Zij] = tube3([X'; Y'; Z']);
Xij = Xij';             %Parametric surface for each of the coordinates,
Yij = Yij';             %where the row corresponds to slices (j), and column
Zij = Zij';             %is the horizontal parameter i.
Xij = Xij(2:end-1, 1:m);            %Remove top and last slices of the tube.
Yij = Yij(2:end-1, 1:m);            %Also, remove the repeated column.
Zij = Zij(2:end-1, 1:m);

%Plot the initial surface and spine together.
drawSpine(X, Y, Z, surfaceAxes, 'n');
drawSurface(Xij, Yij, Zij, surfaceAxes);

%%%%%%%%%%%%%%%%%%%%%%%% Prepare the video recorder %%%%%%%%%%%%%%%%%%%%%%%%
writerObj = VideoWriter('showcase.avi');
open(writerObj);

%%%%%%%%%%%%%%%%%% Setting initial values for surface %%%%%%%%%%%%%%%%%%%%%%
springs = createSurfaceSprings(Xij, Yij, Zij);  %Create the springs.
VXij = zeros(n,m);              %Zero velocity initially.
VYij = zeros(n,m);
VZij = zeros(n,m);
potential = 0;                  %Potential energy associated with the springs.

%%%%%%%%%%%%%%%%%%%% Constants for the applied forces %%%%%%%%%%%%%%%%%%%%%%
b = ones(n,1)*0.35;             %Constant for symmetry force.
b = repmat(b, 1, m);

%%%%%%%%%%%%%%%%%% Values for initializing the simulation %%%%%%%%%%%%%%%%%%
deltaT = 0.001;                 %Time step.
friction = 0.5;                 %Friction term.
sigma = 50.0;                   %Start with a large blurred.
nextSigmaDivision = 1.45;       %Value to divide by each sigma update.

%%%%%%%%%%%%%%%%%%%%%%%% Multiresolution loop %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while(sigma > 1.0)

    %%%%%%%%%% Blurred the image and compute the image gradient %%%%%%%%%%
    gaussImg = gaussianImage(img, sigma);   %Convolve image with a Gaussian.
    magGradImg = computeMagnitudeOfGradientImage(gaussImg); %Mag. of the
                                                            %Grad. Image.

    %Display current image shape to approximate.
    imagesc(magGradImg, 'Parent', imgAxes);

    %Compute gradient from magnitude-of-gradient image (derivatives).
    [Ix Iy] = computeImageGradient(magGradImg);
```

```
%Set a term for linear interpolation.
alpha = sigma/50.0;

%Set the c constant as varying over sigma.
c = zeros(n, 1);                 %Constant for contraction/expansion.
if(sigma > 5.0)
    c = ones(n, 1)*(alpha * 0.00);    %Constant for first half of tube.
    c(floor(n/2)+1:n, 1) = ones(n-floor(n/2), 1)*(alpha * 0.00);
                                      %Constant for second half of tube.
end;
c(1,1) = -0.04;
c(n,1) = -0.04;
c = repmat(c, 1, m);

%Set the spring and damping force constants.
if(sigma > 10.0)
    Ksp = 4.5;
    Kd = 4.5;
else
    Ksp = 7.5;
    Kd = 7.5;
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%% Simulation loop %%%%%%%%%%%%%%%%%%%%%%%%%%%
iteration = 1;                           %Control number of iterations.

%Initialize previous state of the surface.
XijOld = zeros(n, m);
YijOld = zeros(n, m);
ZijOld = zeros(n, m);

%Preliminar speed of the surface.
vXij = XijOld - Xij;
vYij = YijOld - Yij;
vZij = ZijOld - Zij;         %Surface velocity.
diffMag = sqrt(vXij.^2 + vYij.^2 + vZij.^2);
surfaceSpeed = sum(sum(diffMag, 2))/(n*m);  %Average speed.

while(surfaceSpeed > 0.001 && iteration <= 100)   %Update Xij, Yij, and
                                                  %Zij in the same loop.

    %%%%%%%%%%%%%%%%%%%% Computation for the surface %%%%%%%%%%%%%%%%%%%%
    XijOld = Xij;            %Old values for surface.
    YijOld = Yij;
    ZijOld = Zij;

    %Compute the spring forces.
    if(potential >= 5.05 && sigma > 10)  %Relax the springs attached to
                                         %the surface.
        fprintf('\nRelaxing springs, wait...');
        springs = relaxSurfaceSprings(springs, Xij, Yij, Zij);
        fprintf('\nSurface springs have been relaxed!');
    end;
    [FspX FspY FspZ potential] = computeSurfaceSpringForce(springs,...
         Xij, Yij, Zij, VXij, VYij, VZij, Ksp, Kd);
```

```
        %Computing the symmetry-around-the-spine force. Force B.
        [FtBx FtBy FtBz unitRadiiStructure] = get_forceB(Xij, Yij, Zij, X,...
            Y, Z, b);

        %Computing the expansion/contraction force. Force C.
        [FtCx FtCy FtCz] = get_forceC(Xij, Yij, Zij, X, Y, Z, c,...
            unitRadiiStructure);

        %Compute the image forces on the surface.
        [FimgX FimgY] = computeSurfaceImageForce(Xij, Yij, Zij, Ix, Iy,...
            height, width, unitRadiiStructure);

        %%%%%%%%%%%%%%%%%%%%%%%%% Update the surface %%%%%%%%%%%%%%%%%%%%%%
        %First X.
        VXij = VXij + deltaT/friction * FspX + FimgX + FtBx + FtCx;
        clampPos = find(abs(VXij) > 1.0);
        VXij(clampPos) = 1.0 * sign(VXij(clampPos));
        Xij = XijOld + friction/5.0 * VXij;

        %Y component.
        VYij = VYij + deltaT/friction * FspY + FimgY + FtBy + FtCy;
        clampPos = find(abs(VYij) > 1.0);
        VYij(clampPos) = 1.0 * sign(VYij(clampPos));
        Yij = YijOld + friction/5.0 * VYij;

        %Z component.
        VZij = VZij + deltaT/friction * FspZ + FtBz + FtCz;
        clampPos = find(abs(VZij) > 1.0);
        VZij(clampPos) = 1.0 * sign(VZij(clampPos));
        Zij = ZijOld + friction/5.0 * VZij;

        %Recompute surface speed.
        diffMag = sqrt(VXij.^2 + VYij.^2 + VZij.^2);
        surfaceSpeed = sum(sum(diffMag, 2))/(n*m);  %Average speed.

        %%%%%%%% Draw spine and surface in their new configuration %%%%%%%%
        drawSpine(X, Y, Z, surfaceAxes, 'n');
        drawSurface(Xij, Yij, Zij, surfaceAxes);
        frame = getframe;                   %Write the frame in the video.
        writeVideo(writerObj,frame);
        drawnow;

        %%%%%%%%%%%%%%%%%%%%%%% Display information %%%%%%%%%%%%%%%%%%%%%%%
        fprintf('\nSigma= %f, Iter= %i, Surf Sp= %f, Potential= %f',...
            sigma, iteration, surfaceSpeed, potential);

        iteration = iteration + 1;     %Next iteration.
    end;

    nextSigmaDivision = nextSigmaDivision + 0.03;
    sigma = sigma / nextSigmaDivision;    %Decrease sigma for next shape
                                          %approximation.
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Rotating the surface %%%%%%%%%%%%%%%%%%%%%%%%%%
cla(surfaceAxes);
img = imread('bananatexture.jpg');
```

```
surface([Xij, Xij(:,1)], [Yij, Yij(:,1)], [Zij, Zij(:,1)], img,...
    'Parent', surfaceAxes, 'EdgeColor', 'none', 'FaceColor', 'texturemap',...
    'Marker', 'none');
camlight;
lighting phong;
axis vis3d;
oh=findobj(gca,'type','surface');
for I = 1:360
    rotate(oh,[0 1 0],1);
    frame = getframe;                          %Write the frame in the video.
    writeVideo(writerObj,frame);
    drawnow;
end;

close(writerObj);                              %Close the video recorder.
```

Function to Draw the Spine

```
function drawSpine(X, Y, Z, axesHandle, closed)
%Function to draw a spine given by points in 'X', 'Y', 'Z', over the
%axesHandle, and closed or not closed (connecting last with initial point).

if(closed == 'c')                                  %Close loop.
    plot3([X; X(1)], [Y; Y(1)], [Z; Z(1)],...
        'Color', [1.0 0.65 0], 'LineWidth', 2,...
        'Marker', '.', 'MarkerSize', 9, 'MarkerEdgeColor', [1 1 0],...
        'Parent', axesHandle);
else
    plot3(X, Y, Z, 'Color', [1.0 0.65 0], 'LineWidth', 2,...
        'Marker', '.', 'MarkerSize', 9, 'MarkerEdgeColor', [1 1 0],...
        'Parent', axesHandle);
end;
```

Function to Draw the Surface

```
function drawSurface(Xij, Yij, Zij, axesHandle)
% Function to plot the surface given by its grid parametric space 'Xij',
% 'Yij', and 'Zij' coordinates.
surface([Xij, Xij(:,1)], [Yij, Yij(:,1)], [Zij, Zij(:,1)],...  %Repeat the
                                                    %first and last column.
    'Parent', axesHandle, 'EdgeColor', 'none', 'FaceColor', [0.9 0.8 0.0],...
    'Marker', '.', 'MarkerSize', 2, 'MarkerEdgeColor', [1 0 0]);
camlight;
lighting phong;
```

Function to Create the Surface Springs

```
function springs = createSurfaceSprings(Xij, Yij, Zij)
% Function to create the springs that define the surface deformation
% forces. 'Xij', 'Yij', and 'Zij' are 2D matrices that contain the
% coordinates of each node on the parametric grid that defines the surface.
% 'springs' is a 1D matrix that contains rows that define the rest length
% and the parametric coordinates (i,j) of two particles connected with such
% spring.
global n m;                     %Make use of the global dimensions.
totalSprings = n*m + (n-1)*m; %Consider both horizontal and vertical springs.
springs = zeros(totalSprings, 5);   %Initialize the springs structure.
```

```
springIndex = 1;

%First, create the horizontal springs.
for J = 1:n                          %Rows first.
    [~, D] = computeLength(Xij(J,:)', Yij(J,:)', Zij(J,:)', 'c');  %Get the
                                 %average distance between points in one row.
    avgD = mean(D);
    for I = 1:m                      %Columns then.
        iMinus1 = adjust1DIndex(m, I-1, 'c');   %It is a closed loop along x.
        springRow = [avgD,      I,      J,      iMinus1,    J];
                    %restlength p1x     p1s     p2x         p2s -> components
                                                            %of the spring.
        springs(springIndex,:) = springRow;
        springIndex = springIndex + 1;
    end;
end;

%Then, create the vertical springs.
for I = 1:m                          %Columns first.
    for J = 2:n                      %Rows then.
        vDistance = [Xij(J,I) Yij(J,I) Zij(J,I)] - [Xij(J-1,I) Yij(J-1,I) Zij(J-
1,I)];
        distance = norm(vDistance);
        springRow = [distance, I, J, I, J-1];   %-> components of the spring.
        springs(springIndex,:) = springRow;
        springIndex = springIndex + 1;
    end;
end;
```

Function to Compute the Length of a String of Particles

```
function [L, D] = computeLength(X, Y, Z, closed)
% Function to compute the length of a curve given by its points in 3D
% coordinates: 'X', 'Y', and 'Z'. The result is stored in a column vector
% 'L', with the cumulative distance from X1 to Xj. Vector 'D' stores the
% distances between jth point with respect to (j-1)th point. The closed
% parameter, when it is "c", indicates that last point and first one
% will be connected.
n = size(X, 1);                 %Take X, Y, and Z as column vectors.
L = zeros(n, 1);
D = zeros(n, 1);
for I = 2:n
    vectorDistance = [X(I) Y(I) Z(I)] - [X(I-1) Y(I-1) Z(I-1)];
    D(I) = norm(vectorDistance);
    L(I) = L(I-1) + D(I);
end;

%Give first point a distance equal to v(1) - v(n) to close curve.
if(closed == 'c')
    D(1) = norm([X(1) Y(1) Z(1)] - [X(n) Y(n) Z(n)]);
else
    D(1) = norm([X(1) Y(1) Z(1)] - [X(2) Y(2) Z(2)]); %Link first and
                                                      %second point.
end;
```

Function to Validate Circular and Non-Circular Indexes

```
function adjustedIndex = adjust1DIndex(n, J, closed)
```

```
%Function to take an index and return a circular valid index if closed is
%"c", or -1 if closed is different to "c".
if(J <= 0)                  %Below boundary?
    if(closed == 'c')
        J = J + n;          %Join first and last points.
    else
        J = -1;             %Since curve is not closed, return invalid index.
    end;
end;

if(J > n)                   %Above boundary?
    if(closed == 'c')
        J = J - n;          %Join last and first points if curve is closed.
    else
        J = -1;             %Otherwise, return an invalid index.
    end;
end;

adjustedIndex = J;  %Return circular index.
```

Function to Blur an Image with a Gaussian Kernel

```
function blurred = gaussianImage(grayScaleImage, sigma)
% Function that convolves a 2D-matrix 'grayScaleimage' with a Gaussian with
% standard deviation given by 'sigma'. The resulting 'blurred' image is also
% in gray scale but in double values. Input should be a gray-scale picture.

Channel = double(grayScaleImage);
if(isinteger(grayScaleImage))
    Channel = Channel / 255.0;
end;

kernel = fspecial('gaussian', [1, ceil(5 * sigma)*2+1], sigma);
V = conv2(kernel, kernel, Channel, 'same');     %Convolve in both directions.

% Normalize the output.
minValue = min(min(V));
maxValue = max(max(V));
blurred = (V - minValue) / (maxValue - minValue);     %Return 1 channel in
                                                      %double values.
```

Function to Compute the Image Potential Field

```
function gradImage = computeMagnitudeOfGradientImage(grayScaleImage)
% Function to compute the magnitude-of-the-gradient image from a gray scale
% picture given as input. The returning 'gradImage' is also a gray scale
% image.

Channel = double(grayScaleImage);
if(isinteger(grayScaleImage))
    Channel = Channel / 255.0;
end;

[imgGradX imgGradY] = computeImageGradient(Channel);

imgGradMag = sqrt(imgGradX.^2 + imgGradY.^2);   %Magnitude of the gradient.
```

```
% Normalize the output.
[h w] = size(imgGradMag);
imgGradMag(1,:) = 0;
imgGradMag(h,:) = 0;
imgGradMag(:,1) = 0;
imgGradMag(:,w) = 0;
minValue = min(min(imgGradMag));
maxValue = max(max(imgGradMag));
gradImage = (imgGradMag - minValue) / (maxValue - minValue);
```

Function to Compute the Image Gradient

```
function [imgGradX imgGradY] = computeImageGradient(grayScaleImage)
% Function to compute the image gradient in both X and Y from a
% 'grayScaleImage' given as input. The returning images are also gray scale
% images.
% 'imgGradX' and 'imgGradY' are double-valued matrices in the range 0, 1.

Channel = double(grayScaleImage);
if(isinteger(grayScaleImage))
    Channel = Channel / 255.0;
end;

SobelOperator = [1 2 1; 0 0 0; -1 -2 -1]; %Use the Sobel operator for
                                          %gradient.
imgGradX = conv2(Channel, SobelOperator', 'same');    %I sub x (Vertical
                                                      %edges).
imgGradY = -1*conv2(Channel, SobelOperator, 'same');  %I sub y (Horizontal
                                                      %edges).
```

Function to Relax the Surface Springs

```
function springs = relaxSurfaceSprings(springs, Xij, Yij, Zij)
% Function to update the rest length of the surface springs in order to
% relieve the tension. 'springs' is the 2D matrix of springs data; and
% 'Xij', 'Yij', and 'Zij' are the coordinates of the points of the surface.
% The returning 2D 'springs' matrix is an updated version of the one given
% as input.
global n m;                     %Make use of the global dimensions.

springIndex = 1;

%First, update the horizontal springs.
for J = 1:n                       %First rows.
    [~, D] = computeLength(Xij(J,:)', Yij(J,:)', Zij(J,:)', 'c'); %Get the
                                %average distance between points in one row.
    avgD = mean(D);
    newRestLength = springs(springIndex,1)+(avgD-springs(springIndex,1))/2.0;

    for I = 1:m                       %Columns then.
        springs(springIndex,1) = newRestLength; %Assign the row restlength to
                                                %all of the column nodes.
        springIndex = springIndex + 1;
    end;
end;

%Then, update the vertical springs.
```

```
for I = 1:m                          %Columns first.
    for J = 2:n                      %Rows then.
        vDistance = [Xij(J,I) Yij(J,I) Zij(J,I)] - [Xij(J-1,I) Yij(J-1,I)...
            Zij(J-1,I)];
        distance = norm(vDistance);
        if(distance > springs(springIndex,1))   %Did it shrink?
            newRestLength = distance / 1.05;
        else
            newRestLength = springs(springIndex,1)+(distance-...
                  springs(springIndex,1))/2.0;
        end;

        springs(springIndex,1) = newRestLength;

        springIndex = springIndex + 1;
    end;
end;
```

Function to Compute the Spring Force

```
function [FspX FspY FspZ potential] = computeSurfaceSpringForce(springs, Xij, Yij,
Zij, VXij, VYij, VZij, Ksp, Kd)
% Function to compute the spring forces that keep the surface together.
% 'springs' is a 2D matrix of springs defined in createSurfaceSprings.
% 'Xij', 'Yij', and 'Zij' are the coordinates of the 3D surface; and
% 'VXij', 'VYij', and 'VZij' are the velocities of each point on the
% surface. 'Ksp' and 'Kd' are the spring and damping constants
% respectively.
% 'FspX', 'FspY', and 'FspZ' are nxm matrices with the spring force
% components. 'potential' is the average elastic potential energy,
% accumulated as consequence of the spring forces.
global n m;                      %Make use of the global dimensions.
sTotal = size(springs, 1);       %Number of springs in the system.
FspX = zeros(n,m);               %Initialize the force matrices.
FspY = zeros(n,m);
FspZ = zeros(n,m);

%Potential energy of the springs.
potential = 0;

for K = 1:sTotal
    %Obtain data from current spring.
    restLength = springs(K,1);      %Rest length.
    I1 = springs(K,2);              %x coordinate of the first particle.
    J1 = springs(K,3);              %s coordinate of the first particle.
    I2 = springs(K,4);              %x coordinate of the second particle.
    J2 = springs(K,5);              %s coordinate of the second particle.

    %Compute the vector between the two particles.
    vector = [Xij(J1,I1) Yij(J1,I1) Zij(J1,I1)] - [Xij(J2,I2) Yij(J2,I2)...
      Zij(J2,I2)];
    distance = norm(vector);
    unitVector = vector./distance;

    %Compute spring force.
    potential = potential + (restLength - distance)^2.0;
    F1 = Ksp * (restLength - distance) * unitVector;    %For particle 1.
    F2 = -F1;                                           %For particle 2.
```

```
    %Compute damping force.
    velocity = [VXij(J1,I1) VYij(J1,I1) VZij(J1,I1)] - [VXij(J2,I2)...
      VYij(J2,I2) VZij(J2,I2)];
    D1 = -Kd * dot(velocity, unitVector) * unitVector;  %For particle 1.
    D2 = -D1;                                           %For particle 2.

    %Add forces for particle 1.
    total1 = F1 + D1;
    FspX(J1,I1) = FspX(J1,I1) + total1(1);    %X component.
    FspY(J1,I1) = FspY(J1,I1) + total1(2);    %Y component.
    FspZ(J1,I1) = FspZ(J1,I1) + total1(3);    %Z component.

    %Add forces for particle 2.
    total2 = F2 + D2;
    FspX(J2,I2) = FspX(J2,I2) + total2(1);    %X component.
    FspY(J2,I2) = FspY(J2,I2) + total2(2);    %Y component.
    FspZ(J2,I2) = FspZ(J2,I2) + total2(3);    %Z component.
end;

potential = potential * 0.5 * Ksp / sTotal;   %Average potential energy.
```

Function to Compute the Radial-Symmetry Force

```
function [ fbx, fby, fbz, unitRadiiStructure ] = get_forceB( vtx, vty, vtz, vsx,
vsy, vsz, bs )
%Computes the force associated with radial symmetry at each point on the
%tube surface
%   @vt: 3 MxN matrices of tube coordinates corresponding to x, y, and z
%       coordinates, respectively.  M corresponds to the vertical parameter,
%       and N corresponds to the horizontal parameter. (ie: a row represents
%       a circular "slice")
%   @vs: 3 column vectors of spine coordinates (x, y, z coordinates
%       respectively).
%   @bs: controls the strength of the force-can be a single scalar value if
%       it is the same for all
%       points, or a matrix of values corresponding to each point of the tube
%       surface)
%   @fbx, fby, fbz: MxN matrices that represent the force components for
%       each particle in the surface.
%   @unitRadiiStructure: a data structure with the unit radii components
%       for each particle in the surface.
[rx,ry,rz] = get_rad_vec( vtx, vty, vtz, vsx, vsy, vsz );
r_mag = sqrt( (rx.*rx) + (ry.*ry) + (rz.*rz) );
r_mean = get_mean_rad_vec( vtx, vty, vtz, vsx, vsy, vsz );

[ rx_unit, ry_unit, rz_unit ] = get_unit_rad_vec(rx, ry, rz);
r_diff = r_mean - r_mag;

fbx = bs .* (r_diff .* rx_unit);
fby = bs .* (r_diff .* ry_unit);
fbz = bs .* (r_diff .* rz_unit);

%Return also the unit radii to avoid recomputing them in get_forceC.
unitRadiiStructure{1} = rx_unit;
unitRadiiStructure{2} = ry_unit;
unitRadiiStructure{3} = rz_unit;
```

Function to Compute the Radial Vector Function

```
function [ rx, ry, rz ] = get_rad_vec( vtx, vty, vtz, vsx, vsy, vsz )
%Calculates and returns the radial vector function for a given tube and
%spine
%   @vt: 3 MxN matrices of tube coordinates corresponding to x, y, and z
%       coordinates, respectively.  M corresponds to the vertical parameter,
%       and N corresponds to the horizontal parameter. (ie: a row represents
%       a circular "slice")
%   @vs: 3 column vectors of spine coordinates (x, y, z coordinates
%       respectively).
%   @rx, ry, rz: MxN matrices with the radial vector for each point in the
%       tube.

%Build matrix with columns [representing spine coordinates] repeated to
%match size of tube matrices
[~,n] = size(vtx);
vsx1 = kron(vsx,ones(1,n));
vsy1 = kron(vsy,ones(1,n));
vsz1 = kron(vsz,ones(1,n));

%Get radial vector for each point on the tube surface
rx = vtx - vsx1;    %x-coords
ry = vty - vsy1;    %y-coords
rz = vtz - vsz1;    %z-coords
```

Function to Obtain the Mean Radius among Particles in the Same Slice of the Tube

```
function [ r_mean ] = get_mean_rad_vec( vtx, vty, vtz, vsx, vsy, vsz )
%Returns a matrix where elements of the j-th row are the mean radius at
%"level" j (all elements of a row in this matrix are the same).
%   @vtx, vty, vtz:  the x, y, z coordinates of the tube respectively
%   @vsx, vsy, vsz:  the x, y, z coordinates of the spine respectively
%   @r_mean: MxN matrix containing the mean radii for each "slice" of the
%       tube.

%Get magnitude of radius at each point
[ rx, ry, rz ] = get_rad_vec( vtx, vty, vtz, vsx, vsy, vsz );
r_mag = sqrt( (rx.*rx) + (ry.*ry) + (rz.*rz) );

%Compute mean radius for each level
[~,n] = size(vtx);
r_mean = mean(r_mag, 2);
r_mean = repmat(r_mean, 1, n);
```

Function to Obtain the Unit Radial Vector of Each Particle

```
function [ rx_unit, ry_unit, rz_unit ] = get_unit_rad_vec( rx, ry, rz )
%Returns the unit radial vector function
%   @rx, ry, rz, are the x,y,z coordinates (respectively) of the radial
%       vector function
%   @rx_unit, ry_unit, rz_unit: NxM matrices with the unit radii for the
%       points on the tube.

r_mag = sqrt( (rx.*rx) + (ry.*ry) + (rz.*rz) );
rx_unit = rx ./ r_mag;
ry_unit = ry ./ r_mag;
```

```
rz_unit = rz ./ r_mag;
```

Function to Compute the Expansion/Contraction Force

```
function [ fcx fcy fcz ] = get_forceC( vtx, vty, vtz, vsx, vsy, vsz, cs,
unitRadiiStructure )
%Computes the force associated with expansion and contraction
%   @vtx, vty, vtz:  the x, y, z coordinates of the tube respectively
%   @vsx, vsy, vsz:  the x, y, z coordinates of the spine respectively
%   @cs: controls the strength of the force  (can be a single scalar value if
%       it is the same for all points, or a matrix of values corresponding to
%       each point of the tube surface)
%   @unitRadiiStructure is an optional data structure with the nxm matrices
%       that contain the unit radii computed in get_forceB.
%   @fcx, fcy, fcz: MxN matrices with the resulting c force components.

if(nargin == 7)
    [rx, ry, rz] = get_rad_vec( vtx, vty, vtz, vsx, vsy, vsz );
    [ rx_unit, ry_unit, rz_unit ] = get_unit_rad_vec( rx, ry, rz );
end;

if(nargin == 8)
    rx_unit = unitRadiiStructure{1};      %Extract the nxm matrices from the
                                          %unitRadiiStructure.
    ry_unit = unitRadiiStructure{2};
    rz_unit = unitRadiiStructure{3};
end;

fcx = cs .* rx_unit;
fcy = cs .* ry_unit;
fcz = cs .* rz_unit;
```

Function to Compute the Image Force

```
function [FX FY] = computeSurfaceImageForce(Xij, Yij, Zij, imgFieldX, imgFieldY,
height, width, unitRadiiStructure)
% Function to compute the forces that act on the surface points. 'Xij',
% 'Yij', and 'Zij' are nxm matrices with the 3D components of the
% parametric surface. 'imgFieldX' and 'imgFieldY' are the derivatives of
% the image in both directions. 'height' and 'width' are the size of the
% image in pixel values.
% 'FX' and 'FY' are nxm force matrices for each of the X and Y components
% of the parametric surface.
global ImgPlaneNormal n m;  %Use global size and image plane normal matrices.

%First, compute the normals to the surface.
[Xnormal Ynormal Znormal] = computeSurfaceNormals(Xij, Yij, Zij);
P3D(:,:,1) = Xnormal;       %And store the normals in a 3D matrix.
P3D(:,:,2) = Ynormal;
P3D(:,:,3) = Znormal;

%Now determine which points are perpendicular to the image plane.
absDots = abs(dot(P3D, ImgPlaneNormal, 3)); %Dot product along the third
                                            %dimension.
locations = find(absDots < 0.05);           %Pick the locations only where
                                            %the absolute value is less than
                                            %0.05.
```

```
%Initialize the forces for the two components.
FX = zeros(n, m);
FY = zeros(n, m);

%Extract the nxm matrices from the unitRadiiStructure.
rxUnit = unitRadiiStructure{1};
ryUnit = unitRadiiStructure{2};

for I = locations'                %Compute forces only for the locations.
    %Transform coordinates.
    coordinates(1,1) = Xij(I) + width/2.0 + 1.0;
    coordinates(2,1) = -Yij(I) + height/2.0 + 1.0;

    %Compute angle of image force with respect to unit radius in order to
    %avoid uphill motion.
    fX = get_pix_vals(imgFieldX, coordinates(:, 1));
    fY = get_pix_vals(imgFieldY, coordinates(:, 1));
    deltaI = [fX fY];                          %Image gradient vector.
    dotProd = dot(deltaI, [rxUnit(I) ryUnit(I)]);
    angle = acos(dotProd/norm(deltaI));
    if(angle > 0.1765 && angle <  2.9671)      %Angle between 10 and 170°?
        F = dotProd * [rxUnit(I) ryUnit(I)];   %Project image gradient onto
                                               %radius.
    else
        F = deltaI;                            %Use directly the image force.
    end;

    %Assign force.
    FX(I) = (1.0-absDots(I)) * F(1); %Forces from X derivative.
    FY(I) = (1.0-absDots(I)) * F(2); %Forces from Y derivative.

end;
```

Function to Compute the Surface Normal Vectors

```
function [Xnormal Ynormal Znormal] = computeSurfaceNormals(Xij, Yij, Zij)
% Function that computes the normal vector to each node on the surface.
% 'Xij', 'Yij', and 'Zij' are 2D matrices that store the (x,y,z)
% coordinates of each point. 'Xnormal', 'Ynormal', and 'Znormal' store the
% resulting normal vectors, and they have the same dimensions (nxm) of the
% inputs matrices.
global n m;                    %Use the global dimensions.

DXx = zeros(n, m);             %Prestorage derivative direction matrices.
DYx = zeros(n, m);
DZx = zeros(n, m);
DXs = zeros(n, m);
DYs = zeros(n, m);
DZs = zeros(n, m);

for J = 1:n     %Process each row; j is the parameter that varies the least.
    [DX DY DZ] = computeFirstDerivative(Xij(J,:)', Yij(J,:)', Zij(J,:)', 1.0, 'n',
'x');
    DXx(J,:) = DX';    %Direction of the derivative of X with respect to i.
    DYx(J,:) = DY';    %Direction of the derivative of Y with respect to i.
    DZx(J,:) = DZ';    %Direction of the derivative of Z with respect to i.
```

```
end;

for I = 1:m             %Compute the derivatives with respect to j.
    [DXs(:,I) DYs(:,I) DZs(:,I)] = computeFirstDerivative(Xij(:,I),...
            Yij(:,I), Zij(:,I), 1.0, 'n', 's');
end;

% Now compute the normal at each node
Xnormal = DYx.*DZs - DZx.*DYs;  %It computes the normal with a cross-product.
Ynormal = DZx.*DXs - DXx.*DZs;
Znormal = DXx.*DYs - DYx.*DXs;

% Normalizing the resulting vector
Magnitude = sqrt(Xnormal.^2 + Ynormal.^2 + Znormal.^2);
Xnormal = Xnormal./Magnitude;
Ynormal = Ynormal./Magnitude;
Znormal = Znormal./Magnitude;
```

Function to Approximate the First Derivatives with Finite Differences

```
function [DX DY DZ] = computeFirstDerivative(X, Y, Z, delta, closed, withRespectTo)
% Function to compute the first derivative along a univariate, parametric
% function. 'X', 'Y', and 'Z' are column vectors that contain the
% coordinates of the poins along the parametric 1D curve. 'delta' is the
% parametric spacing between nodes. 'closed' indicates if the curve is
% closed. 'withRespectTo' defines if the parametric curve is along the i
% parameter (horizontal), or along the j parameter (vertical).
% 'DX', 'DY', and 'DZ' are column vectors that store the components of the
% first derivative vectors.
global Actrln Actrlm;           %Use the global, central-difference matrices.

if(withRespectTo == 's')        %It is vertically: d/dj.
    DX = 1/delta * Actrln * X;
    DY = 1/delta * Actrln * Y;
    DZ = 1/delta * Actrln * Z;
else                            %It is horizontally: d/di.
    DX = 1/delta * Actrlm * X;
    DY = 1/delta * Actrlm * Y;
    DZ = 1/delta * Actrlm * Z;
end;

if(closed ~= 'c')               %Curve is disconnected?
    DX(1,1) = DX(2,1);          %Copy components of the derivative from the
                                %second point into the first one.
    DY(1,1) = DY(2,1);
    DZ(1,1) = DZ(2,1);
    DX(end,1) = DX(end-1,1);    %Copy components of the derivative from the
                                %last point into the penultimate one.
    DY(end,1) = DY(end-1,1);
    DZ(end,1) = DZ(end-1,1);
end;
```