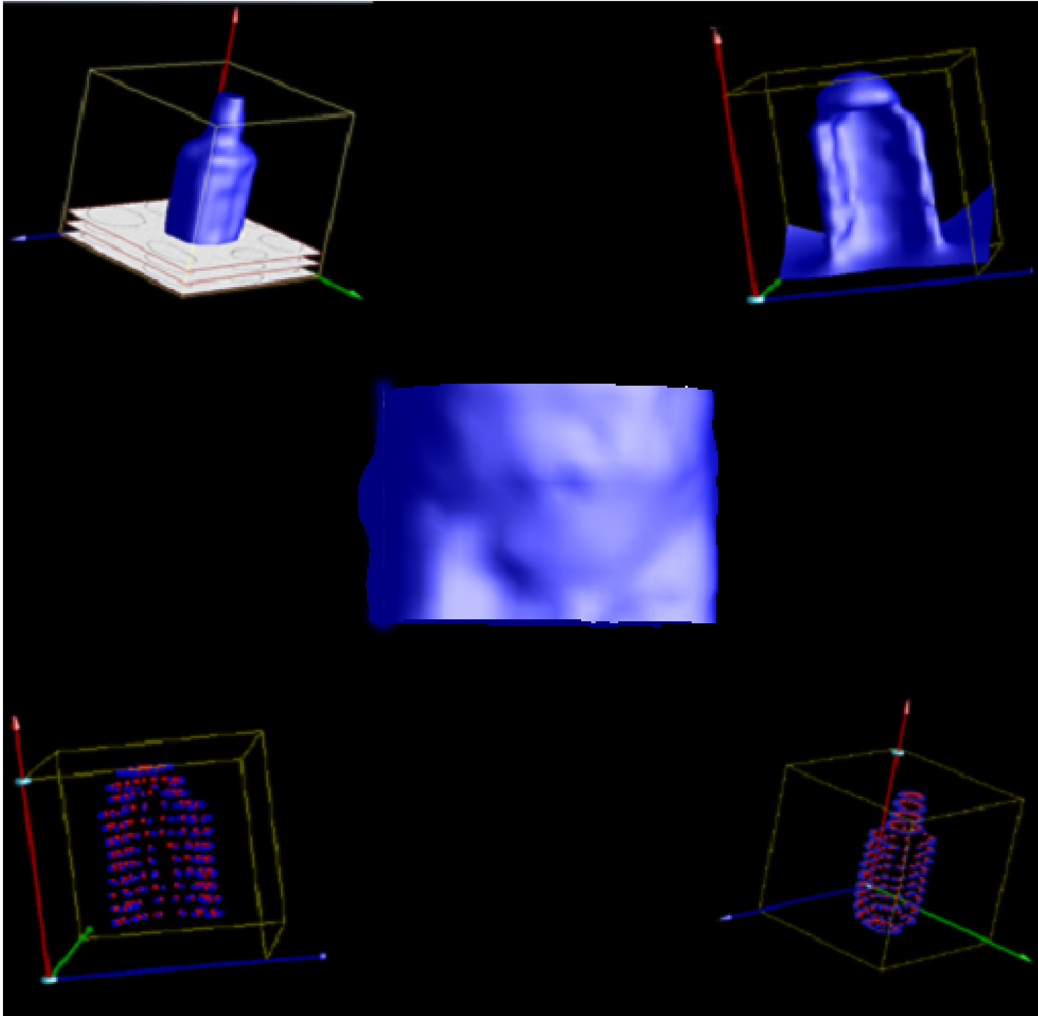
SEMI-AUTOMATIC HIGH-QUALITY RECONSTRUCTION OF THE HEART PATHOLOGY FROM MRI DATA BY DEFORMABLE MESHES



By: Bo Li

Supervisor: Burkhard Wuensche

January 2004

CONTENT

Abstract 3
1. Introduction 3
2. Background 4
    2.1 Introduction of the toolkit from previous program 4
3. Active Contour Models (B-Spline Snakes) 5
    3.1 Introduction to B-Spline Snakes 5
    3.2 The implementation of the B-Spline Snake 7
    3.3 A simple test case for *BSpline* class 13
4. Radial Basic Function 15
    4.1 Introduction to the Radial Basic Function 15
    4.2 Implementation of the Radial Basic Function 17
    4.3 The Analysis and comparison of two different ways to implement The Radial Basic Function in my application 17
5. Sample test case for my application 22
    5.1 Re-arrange the control points to form the closed B-Spline curve 22
    5.2 Test case: Constructing the Pelvis of the human body 23
    5.3 Contained problem and further improvement of my application 26
6. Conclusion and further research 26
Reference 27

Abstract

In this project we investigated and developed a semi-automatic method for the high-quality reconstruction of the heart pathology from MRI data. Traditionally heart pathology is constructed by tracing contours on individual MRI slices by hand. However, for a clinical set-up medical experts require a fast and reliable technique to trace contours automatically or semi-automatically. In a previous project we have developed a tool, which uses a basic soft object model in order to enable users to quickly approximate the heart pathology. In this project we use B-Spline Snakes to adjust the approximate shape obtained with the soft object model so that it will automatically fit the heart contours in every individual MRI data slice. We then reconstruct the heart pathology by using Radial Basic Functions and a Marching Cubes Algorithm. Our results show that the technique offers a fast and convenient way for the effective modelling of organ shapes from medical image data sets.

Keywords: Soft Object Modelling, 3D Marching Cubes, Active Contour Model (B-Spline Snake), Radial Basic Functions.

1. Introduction

In order to construct a high-quality model of the heart shape, medical experts first analyse the heart pathology from every individual MRI data slice and find the exact contour in each of them, and then construct the heart pathology again by fitting a B-Spline surface to the contours taken from MRI data. Currently people in the medical school trace the contour of the heart pathology in each slice by hand, although this is a choice to detect contour on each slices, it's not only very unreliable, but also very time-consuming. As people should spend a lot of time to trace the contour of the heart pathology slice by slice, and if the final model is not look like what it expected to be, the researchers have to trace again by hand, also it increases the difficulty for researcher to determine the reason why the final model is not correct. So a fast and reliable technique is expected to trace the contour of the heart model precisely and constantly.

In order to find a solution to these problems, I firstly use Soft Object Modelling technique to provide some generic Soft Object models in a 3 dimensional environment, so that users can use these models to approximate the shape of the heart combining the generic models (the reason to this is that approximately fitting generic models to the contour is much easier than tracing the contour by hand). After an approximate model of the heart is created, I work out the intersection of the approximate model with the data slices, then I implement the 2 Dimensional Marching Cube to get the interpolate points of the model in each slices. With the help of these interpolate points, I can simply form a closed B-Spline, then I adjust the closed B-Spline to make it exactly fit the contour of the shape at each slices (using B-Spline Snake technique). For the last step, I take a constant number of sample points in every closed splines at each slices and assign different field values to them. With the help of these sample points and

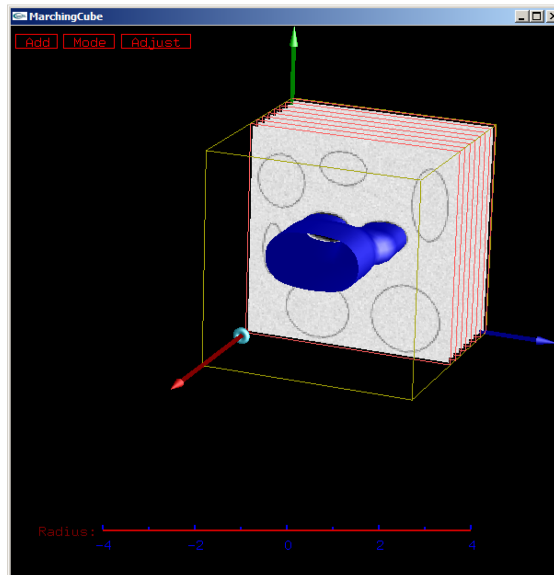
field values, I reconstruct the model of the heart pathology so that its surface goes though all the sample points (using Radial Basic Function and 3D Marching Cubes).
In this report, I will:

- describe the environment and soft object models from my previous project, also some bugs I fit in the previous code and some new function I added to the previous toolkit
- introduce the B-Spline Snake technique for minimising the energy in the spline which can adjust the spline to exactly fit the contour in every image slices
- introduce the Radial Basic Function for creating a 3 dimensional continue field area by using the sample points provided (Combine this result with the Marching Cube can derive a significant smooth 3D model which pass though every sample points)
- describe the problems I met and solved when I was implementing the B-Spline Snake and Radial Basic Function in my program
- introduce some further improvement of the currently project (For example using deformable meshes instead of Radial Basic Function to represent the adjusted closed B-Splines)

2. Background

2.1 Introduction of the toolkit from previous program

The definition of Soft Object Modelling technique is that we define Soft Object models as iso-surface of a density field, which is defined as the sum of distance functions to samples' geometric primitive. A surface is implied by taking an iso-surface through this density field - the higher the iso-surface value, the nearer it will be to the skeleton.
In the previous project, I have implemented the Soft Object Modelling technique to create some generic models in a 3D environment, e.g. Generic Sphere (Soft Ball), generic cylinder (Soft Line), generic cone (Soft Cone), and generic curve (Soft Curve). I have also added some useful functions to the toolkit so that users can simply control and adjust the generic models by using mouse and keyboard, for example changing the radius of the soft object model, adjusting the position of the skeleton of the soft object model, and so on and so force.
The user interface of the program created in my previous project is shown in the Picture 2.1.1.



Picture 2.1.1: Program from previous project

According to the picture of my previous program in picture 2.1.1:

- The red bar in the bottom is used to adjust the size of the selected soft object. For example, adjust the radius of the soft sphere.
- The `Add` button in the top left is a button which will pop up a menu, where user can select the soft object model from.
- The `Modify` button next to the `Add` button is the button to switch the display mode of the program.
- The `Adjust` button is used to modify the selected soft object. For example, start the process which uses B-Spline Snakes and The Radial Basic Functions to reconstruct the model.

More detail about the control of the program I made can be found in my previous report [5].

3. Active Contour Models (B-Spline Snakes)

3.1 Introduction to B-Spline Snakes

This section presents an overview of one popular active contour model: B-Spline Snakes, which is very useful for detecting and tracking the region in an image. Snakes were first introduced in 1987 by Kass [1] et al, and have become popular since then. Technically, a Snake is an energy-minimizing spline guided by external constraint forces and influenced by image forces that pull it toward features such as lines and edges. With the help of this energy-minimizing spline, people can successfully use snakes for interactive interpretation, in which user-imposed constraint force guide the snake near features of interest. Snakes can also be successfully used to solve some problems including detection of edges, lines, and subjective contours, and so on. In this project, I use B-Spline Snakes

to detect the contour of the heart pathology on each slice, and accurately allocate the closed B-Spline on the contour. In the rest of this section, I will briefly introduce the theory of the B-Spline Snake, and in next section I will explain the steps I implement B-Spline Snake and problems I met when I was implementing it.
As we mentioned above: *a Snake is an energy-minimizing spline guided by external constraint forces and influenced by image forces that pull it toward features such as lines and edges*, we need a energy function to make the snakes towards the lines and edges, so the energy function of the B-Spline Snake can be made up of 2 parts: internal constraint forces, external constraint force.
The internal constraint forces of the spline serve to impose a piecewise smoothness constraint, it depends on the intrinsic properties of the spline, and normally it is the sum of the elastic energy and bending energy of the spline. Minimizing both the elastic energy and bending energy can lead to a smoothed spline.
The external constraint forces are responsible for putting the snake near the desired location. These forces are derived from the image data itself. In this project, I use the intensity gradient at each pixel as the main extrinsic force of the B-Spline Snake, but the original image data should be smoothed by Gaussian filter first. The reason for importing the Gaussian filter to the external force is that: the data in the original images are always discontinuous, if part of snakes finds a low-energy image feature, the spline term can not pull the neighbouring parts of the snake toward a possible continuation of the feature. So if we spatially smoothing the image data by Gaussian filter to make it continuous, the whole snake will toward to the low-energy feature of the image.
According to the description above, we represent the position of the snake parametrically by $v(s) = (x(s), y(s))$, then the energy function of the B-Spline Snakes is:
For internal constraint forces:

$$E_{elastic} = \frac{1}{2}\int_{s} \alpha(s)\,|\,v_s\,|^2\,ds \qquad E_{bending} = \frac{1}{2}\int_{s} \beta(s)\,|\,v_{ss}\,|^2\,ds$$

$$E_{\text{int}} = E_{elastic} + E_{bending} = \int_{s} \frac{1}{2}(\alpha\,|\,v_s\,|^2 + \beta\,|\,v_{ss}\,|^2)ds$$

Formula 3.1.1: the equation for calculating internal force in the Snake

Note: both α(s) and β(s) here are the weights of the elastic force and the bending force in the internal energy. Adjusting the weights α(s) and β(s) controls the relative importance of the elastic force and the bending force.

For external constraint forces:

$$E_{ext} = \int_{s} E_{image}(v(s))ds \qquad E_{image}(x,y) = -\,|\,\nabla(G_{\sigma}(x,y) * I(x,y))\,|^2$$

Formula 3.1.2: the equation for calculating bending force in the Snake

The final energy function is:

$$E_{snake} = E_{\text{int}ernal} + E_{external} + E_{constra\,\text{int}}$$

Formula 3.1.3: Snake's energy function

Note: here is also a weight for external forces, adjusting that weight control the relative importance of the external forces in the total energy of the snake.

3.2 The implementation of the B-Spline Snake

According to the B-Spline Snake introduction above, I will describe how I implement the Snake in my project, what problems I met when I was importing it, and how I solved these problems in the following words.
The B-Spline Snakes technology I introduced so far is still in the part of energy function, this is not enough, because working out the energy of the current status of the B-Spline doesn't mean that we make the B-Spline reach the minimum energy feature of the image, In order to implementation the B-Spline Snakes technology, not only I need to correctly implement the energy function (described above), I also need to implement the adjusting method of the B-Spline. The Adjusting method is the method my program used to make the original B-Spline regularly moved toward the edges in the image, this method is very inconsistent, different people can implement it differently, but the main idea is that my program:

- regularly adjust the control points of the B-Spline
- work out the energy of the adjusted status of the B-Spline
- compare the current energy with previous energy, if current energy is less, start from the current status and keep on adjusting, otherwise, do the adjusting again from previous status

These 3 steps will be kept on going until the minimum energy status is found.
I will introduce my implementation of the energy function and adjusting method in the following, and at last I will also provide a simple test case to display how efficient my implementation is.
I create a class called *BSpline* to implement all the basic functions a B-Spline Snake should have. For a simple B-Spline Snake, it should have some functions for importing the initial data, for example the initial control points of the B-Spline and the sample image data the Snake going to look at. A Snake also need some constant functions to work out its total energy and automatically adjusting itself to minimize the total energy. Finally, the Snake should also need to return the sample points in the adjusted B-Spline back, so that they can be used to construct the model. The general structure of the *BSpline* class is in the following picture:

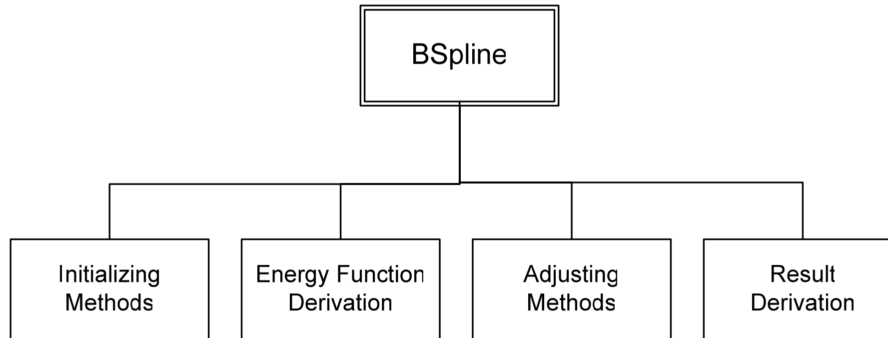


Figure 3.2.1: The Structure of the *BSpline* Class

From *Figure 3.2.1* above, we can notice that a BSpline instance is made up of 4 parts: Initializing Methods, Energy Function Derivation, Adjusting Methods and Result Derivation.

Initializing Methods are some methods for importing the control points of the initial B-Spline and the path of the image data from current directory. These methods are called whenever a new instance of BSpline is created. The control points will be stored in a constant variable of the class, and the image data should be smoothed by a Gaussian filter first before stored into the instance.

Energy Function Derivation and Adjusting methods are used to implement the energy function of the Snake and adjust the B-Spline depend on the total energy, these two parts will be explained in detail in section 3.2.1 and section 3.2.2 in the following.

The main reason why using *BSpline* class is to use it to adjust the initial B-Spline and make it fit the edges in the image, then return the final B-Spline back. So the Result Derivation processes are used to work out the sample points of the B-Spline, a collection of these sample points in 3D can be used to construct the accurate model describe in image data slices (in my program, the result derivation process is going to choose 20 sample points on each B-Spline, explained in section 3.2.3).

3.2.1 The implementation of the energy function

The `getEnergy()` method which calling the `getInternal()` and `getExternal()` methods is the implementation of the energy function in the *BSpline* class. In *BSpline* class, I assume the initial curve is a uniform B-Spline curve, so the parametric equation of B-Spline curve is:

$$P(t) = \sum_{i=0}^{n-1} P_i B_4(t - i + 3) \quad \text{With t in range [0,n-3]}$$

$$B_4(t) = \frac{1}{6}\begin{cases} t*t*t & 0 \le t < 1 \\ v(2-t) & 1 \le t < 2 \\ v(t-2) & 2 \le t < 3 \\ (4-t)3 & 3 \le t < 4 \end{cases}$$

$$\text{where} \quad v(s) = (3s*s*s - 6s*s + 4)$$

Formula 3.2.1: The equation of uniform B-Spline curve

According to the formulas of the internal constraint forces in previous section, the way I work out the integration of elastic force and bending force in the Snake is by dividing the whole curve into a constant number of curve segments depending on the parameter t in the formula above. So the elastic force is the integration of elastic force in each curve segment which can be worked out by inputting the length of the curve into the elastic force formula. For the bending force is a bit tricky, because it requires the first derivative of the uniform B-Spline curve above other than the formula itself. So I pre-calculate the first derivative of uniform B-Spline by hand and hard code it in the *BSpline* class, the first derivation formula is in the following:

$$P(t) = \sum_{i=0}^{n-1} P_i B_4^{'}(t - i + 3) \quad \text{With t in range [0,n-3]}$$

$$B_4^{'}(t) = \frac{1}{6}\begin{cases} 3*t^2 & 0 \le t < 1 \\ -v(2-t) & 1 \le t < 2 \\ v(t-2) & 2 \le t < 3 \\ -3*(4-t)^2 & 3 \le t < 4 \end{cases}$$

$$\text{where} \quad v(s) = (9s^2 - 12s)$$

Formula 3.2.2: The first derivative of uniform B-Spline curve

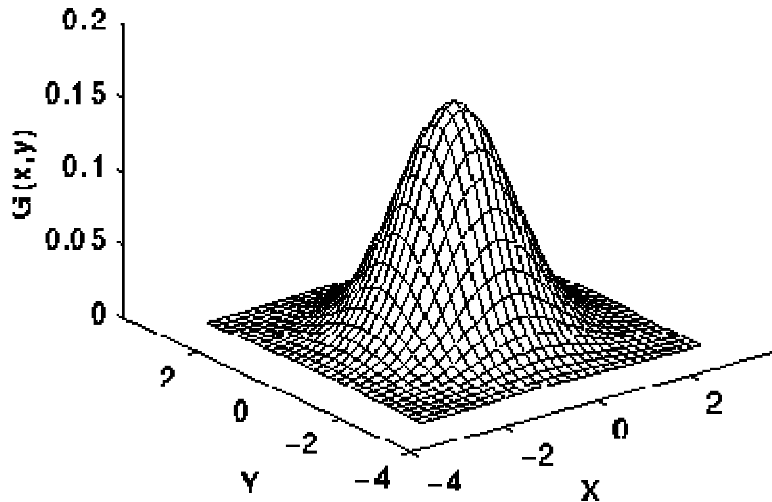
For calculating external constraint forces of the Snake, I first use Gaussian function to smooth the image data and then I compute the intensity gradient of the image data. The higher the gradient is, the smaller the external energy forces of the Snake.

The Gaussian filter is defined as:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Formula 3.2.3: 2D Gaussian Function

The distribution of this Gaussian filter is shown in Picture 3.2.1.1



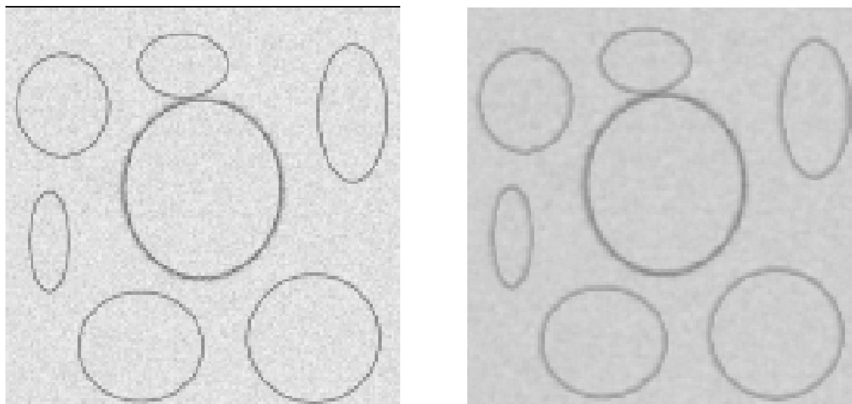
Picture 3.2.1.1: The distribution of 2D Gaussian filter

The smoothing process with Gaussian filter is though a 3 by 3 matrix, so applying this matrix to every pixel of the image data will obviously smoothed image data. The 3 by 3 matrix from Gaussian filter is defined as following

$$\begin{pmatrix} 0.329, 0.199, 0.044 \\ 0.199, 0.121, 0.027 \\ 0.044, 0.027, 0.006 \end{pmatrix}$$

Figure 3.2.1.1: The 3 by 3 matrix created by Gaussian filter

And the pictures before and after the Gaussian filter are illustrated in the picture 3.2.2:



Picture 3.2.1.2: Pictures before and after Gaussian filter

So we can find that using this smoothed image to calculate the intensity gradient of the image is more accurate than the initial image, because this can simply deal with some annoying problems caused by some noise in the image like the picture on the left above. For the noisy image above, if we skip the step of smoothing it by Gaussian filter and do B-Spline Snake directly, the final curve of the Snakes can not fit the edges in the image, as they are not lead towards it.

With the smoothed image data (picture on the right), I simply work out the intensity gradient for a point, says (x_0, y_0), by summing the pixel value difference between point (x_0, y_0) and points around it, (in here I take 4 points which are in just left, right, top and bottom of the point (x_0, y_0)). The formula for work out the intensity gradient is in figure 3.2.1.2:

$$I(x_n, y_n) = \max(v(x_{n-1}, y_n), v(x_n, y_{n-1}), v(x_n, y_{n+1}), v(x_{n+1}, y_n))$$

$$\text{where} \quad v(x, y) = \sqrt{(x - x_n)^2 + (y - y_n)^2}$$

Figure 3.2.1.2: the formula for computing intensity gradient

After worked out the internal constraint forces and external constraint forces, I work out the total energy by sum them up with weights, the weight for internal forces in my program is 100, and the weight for external forces is -0.0002.

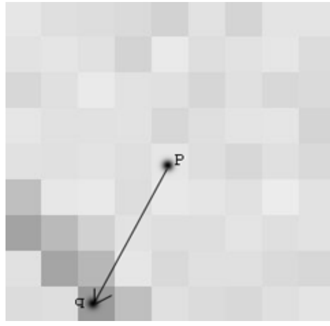
3.2.2 The implementation of the Adjusting Methods

The purpose of adjusting the original B-Spline is to make the spline track the edges in the image so that the minimum energy of the spline is achieved. The basic processes of doing that is keeping on changing the position of every one of the control points which define the B-Spline until the minimum energy spline is achieved. Because the minimum energy of the spline in the image is unknown and the direction of moving each control point is totally random, all these conditions increase the variability and difficulty of fitting the original B-Spline curve into the edges. So a constant and efficient solution to make the original curve trend to the edges is required.

The main idea of my adjusting method is that I apply a move to control point one after another, because changing position of one control point in B-Spline can lead to a change of other parts of the curve, these moves are put into a loop until the minimum energy of the curve is achieved. The following paragraph describe the adjusting method in detail.

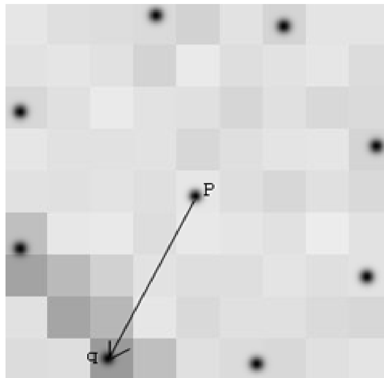
Applying a correct move to a control point is really random and complex, as I need to specified the direction of the move and the distance of the move, also although the control point is in the position of the high intensity gradient, the curve may not pass though the edges we expected in image. So I could only apply a move of the control point to trend to the position we expected each time.

I start defining a move for a specified control point by finding the approximate direction and distance of the move. This process is implemented in the `getClosestStep(CVec3df p)` method. In this method, I wrote a program to find the position of the point p in the image first, then I started a search in a 9*9 square area with position p in the exactly centre. This search is in order to find out the highest gradient point in this area, says q. Finally this method will return a `SPoint` class which is the vector from point p to q. With the help of this `SPoint` instance, I can work out the initial distance by taking the length of the vector and initial direction by taking the direction of this vector. Following picture describe the process of `getClosestStep(CVec3df p)` method:



Picture 3.2.2.1: Picture of deciding the initial info for a move

After I get the `SPoint` instance from `getClosestStep(CVec3df p)` method, I still can not use its length and direction as the content of a move. Because whether point q from image can represent the minimum energy point of previous 9*9 square areas can be determined only depending on the whether the energy of the curve is minimum compared with some other points in this area. So I take 7 other points which can form a circle with q, whose centre is p. as shown in figure 3.2.2.2:



Picture 3.2.2.2: Picture of deciding the initial info for a move

After working out the 7 other positions, I assign the specified control point mentioned before to these points and point p (in total 9 positions) one by one and work out the energy of the curve for each one of them. Finally, the position with lowest energy is the position of the move for that specified point.

The processes of deciding a move mentioned above will be apply to every control points in the curve one after another. In order to make sure the curve created after moving all the control points is the curve with minimum energy, I put the whole processes before into a loop, which will stop until all the move positions for the control points are themselves (this mean the control points don't need to be changed).

3.2.3 Result Derivation processes

Although the Result Derivation processes looks very simple (as it just takes a constant sample points from the final curve, assign field values to them, and return them back), it's quite complex when I am implementing it. There are basically 2 problems:

1. The distance between every 2 sample points is supposed to be constant. This is quite hard to achieve, because it's quite hard to get a constant number of points in the B-Spline with constant distance between them. Even if we get these points, it's quite hard to work out the distance between them on the curve as well.
2. After I get a number of sample points from the curve and assign the field value of them to 0. However, zero-valued sample points are not enough to create subsequent surface for fitting process by Radial Basic Functions (mentioned in the following), I also need positive-valued and negative-valued sample points. For each sample point, I create 2 other points, the line formed by which is perpendicular to the curve on the position of corresponding sample point. After I get these 2 points for each sample points, I should decide what field values I need to assign to them, positive or negative. This is the second problem. As I want to assign positive value to the point inside the closed curve, and negative value to the point outside the closed curve.

The way I deal with the problem 1 is by specifying the value of parameter t to the B-Spline in sequence, and the difference between every continue t is the same. Although this way of taking sample points doesn't help too much, it can still create a list of sample points on the edges in sequence, which can be used to identifying the contour of the model.

There are basically two ways I attempt to solve problem 2:

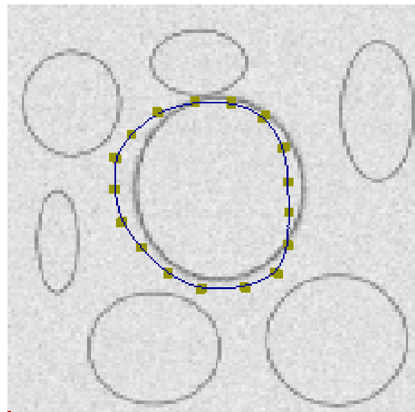
- Method one, I work out the vector which is perpendicular to the curve at the sample point, and then determine two points which correspond to this sample point by adding and subtracting the vector from the sample point. While applying field values to them, I separate this case into situations, for example, if the sample points is taken from clockwise, then the point on its right is assigned to positive and the point on its left is assigned to negative. This way of assigning field values to sample points worked very well for all my test cases, but I am not sure whether it will still work if the case get much more complex.
- Method 2, I simply use the points on the skeleton of the initial soft objects as positive sample points. The benefits of this method are:
    - o Directly derives the positive sample points from the skeleton of the soft objects other than starting a search to detect other sample points.
    - o Produce less sample points than before, more efficient.

The comparison of the above two methods will be shown in detail in section 4.3.

3.3 A simple test case for *BSpline* class

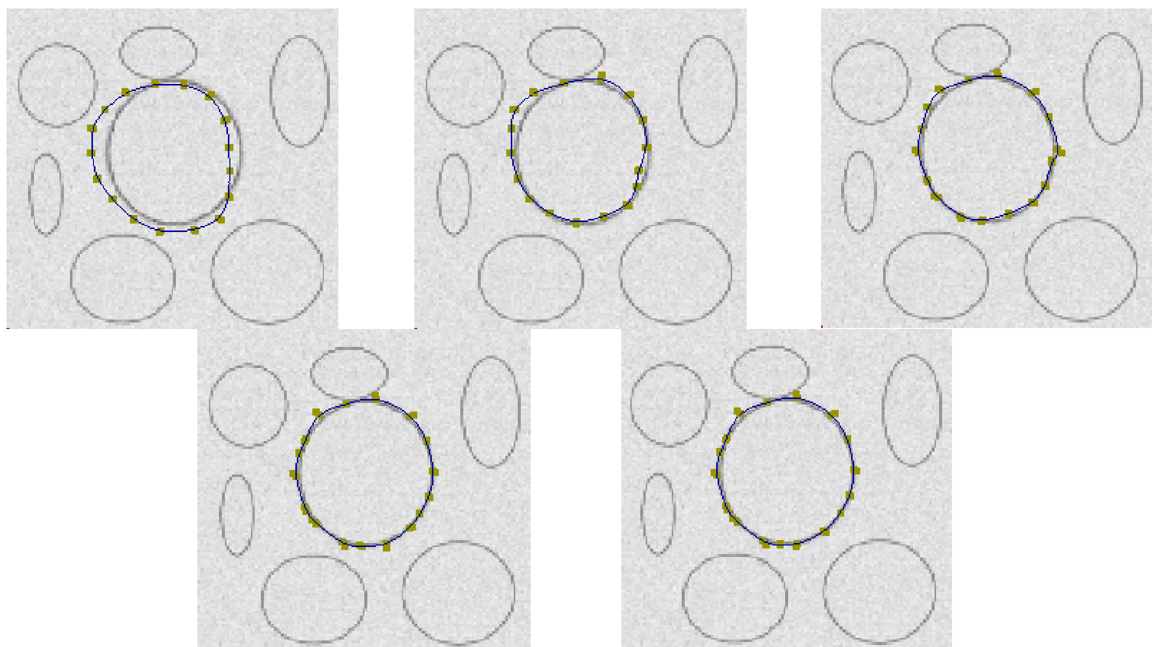
In order to check the performance of the *BSpline* class, I have created a small program which acquires a *BSpline* instance to adjust the original B-Spline curve and make it detect the edges of an ellipse in the middle of the image. Because I just want to test how efficient and effective *BSpline* class is, so one initial curve and one image is enough. In this small program, the original B-Spline curve is

formed by 17 control points and the position of them are defined by me, so that I can make sure the initial curve is far away enough for checking. The original B-Spline curve and image data in this small program are in the following picture:



Picture 3.3.1: The initial curve and image data of the test case

I have also modified the some parts of the codes in this small program compared with the final program I handed in. Other than asking the adjusting processes to keep on running until the final curve is achieved, I assign this task to the “j” button in the keyboard, so every time users click the “j” button, it will adjust every control point only once, users will get the final curve after several attempts of the “j” button. With the help of this “j” button, I can clearly find out how the initial curve be adjusted to match the edges of the ellipse in the image. The process of these adjusting is in the following pictures:

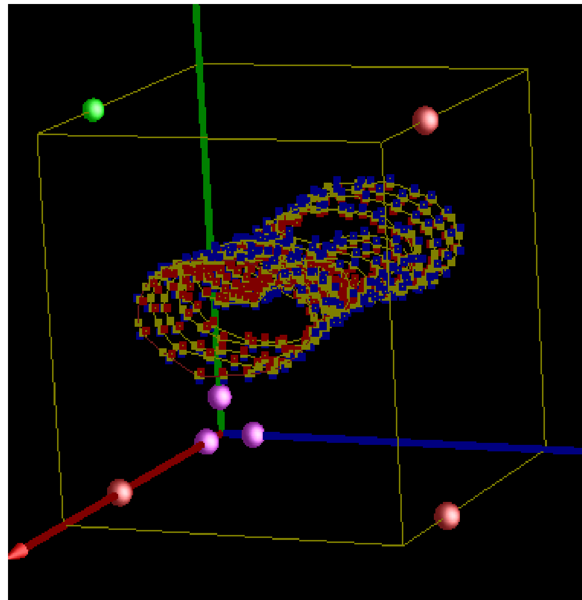


Picture 3.3.2: Processes of adjusting the initial curve to match the edges

With the pictures above, we can notice that the final curve is in the middle of the image and matches the contour precisely.

4. Radial Basic Function

Until now, with the techniques mentioned before, I can precisely get the sample points in the edges of each data slices together with some field values. Like the picture below:



Picture 4.1: Sample Points from data slices

Note: In the above picture, the yellow points are sample points directly taken from the final curve of each data slices and their field values are zero, red points represent points with positive field values, blue points represent points with negative field values.

With this set of zero-valued points and non-zero-valued points, I need to construct a 3D model, whose surface will pass though all the zero-valued points. This raises a problem to interpolate a set of scattered data.

In this section, I will presents an overview of a popular interpolation and reconstruction method from the field of scientific visualization – Radial Basic Function, and I will show how Radial Basic Function can be applied to solve the problem of interpolating scattered sample points in my application. In the following parts of this section, I will:

- briefly introduce the Radial Basic Function theory
- introduce how I implement it in my program
- introduce a sample test case and some contained problems currently in my program of this part

4.1 Introduction to the Radial Basic Function

Radial Basic Functions are used to reconstruct a continue function $f(x)$ from a set of pairs of points and field value pairs, Like {(X_1, f_1), (X_2, f_2), (X_3, f_3), (X_4, f_4)… (X_n,

$f_n)\}$, where X_i represents the position of the sample point in 3D, and f_i represents the field value corresponding to that sample point. The 3D interpolation of the field at the given sample points using Radial Basic Functions is defined as:

$$f(x) = \sum_{i=1}^{n} \alpha_i R(d_i(x)) + p_m(x)$$

Where $$p_m(x) = \sum_{j=1}^{m} \beta_j p_j(x)$$

Formula 4.1.1: Scattered data interpolating method

For the formula above, the basis function $R\ (d_i(x))$ is a non-negative function with the distance function $d_i(x)$ as the input, which represents the distance between point x and the sample point X_i .For my application, I choose The Radial Basic function

$$R(r_i) = r_i * r_i * \log r_i$$

Formula 4.1.2: The definition of Radial Basic Function

For the polynomial part of the interpolating method above, it defines some polynomial basic of degree at most *m*, where variable *m* is a fixed value determined by the number of the sample points I take from previous parts. In my application, I specified the polynomial part in the form of $p(x) = c_1 + c_2x + c_3y + c_4z$.
With the formula mentioned, sample points and sample field values above, we can substitute them into the formula $f\ (x_i) = f_i$ in order to work out the coefficients of the Radial Basic Function.
For the purpose of better understanding the interpolating method above, I can also write the formula in matrix form as follows:

$$\begin{pmatrix} A & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ c \end{pmatrix} = B \begin{pmatrix} \lambda \\ c \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}$$

Formula 4.1.3: Interpolating method in matrix format

Where

- A is an $n*n$ matrix with n equals to the number of the sample points. A_{ij} = R (d_i (x_i)), where d_i (x_i) represents the distance from point *i* in the sample points to point *j*.
- P is a matrix with n rows of (1, x_i, y_i, z_i).
- And λ and c are matrixes of coefficients which I need to work out before I do the interpolating processes.

So after substituting the sample points and sample field values into the matrix above, it will lead to linear system of n+m equations, and I can simply work out the values of the coefficients by using the LUSolver[4].

The LUSolver [4] is a tool which implements the LU decomposition algorithm in order to work out the coefficient part of the Radial Basic Function. LU decomposition is a procedure to decomposing an N*N matrix $\mathbf{A}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$:

$$\mathbf{LU} = \mathbf{A}.$$

The introduction of LU decomposition can be found in [4].

4.2 Implementation of the Radial Basic Function

The procedure of analyzing the sample points on the edges and creating a 3D model passing though all the sample points is getting simpler if I split this task into 4 steps in my application:

- Write an algorithm, which will work out the distance between every 2 sample points and form the *B* matrix part in formula 4.1.3.
- Use the *B* matrix and a matrix of field values to form the Radial Basic Function equation.
- Use LUSolver algorithm to work out the coefficient of the equation (U part of the LUSolver).
- Write an algorithm which recalculates all the field value in the working environment by Radial Basic Function, and store these field values into a float array.

In my application, I combine the above 4 steps all together into a class called *LUSolver*. With the help of this *LUSolver* class, I can do all these 4 important processes though a simple method call. So after user worked out the sample points in each adjusted B-Spline curve, they assign field values to every sample points, and pass all information above in to a method called *display* in the *LUSolver class*, then a new fielded cube will be returned from the method. The performance of these *LUSolver class* will be shown in the following section with test cases.

4.3 The Analysis and comparison of two different ways to implement The Radial Basic Function in my application

The most difficult part while implementing the Radial Basic Function (even for my whole application) is the way to work out positive and negative sample points after the adjusted curves are given. Because sample points in the adjust curves are all zero-valued points, and these are far not enough to determine the field values in the whole working cube by Radial Basic Function. So we need a serial of valid positive and negative sample points to help define the field values of the whole cube. This raises a problem of deciding how many points that we going to use and which way to determine positive and negative sample points.

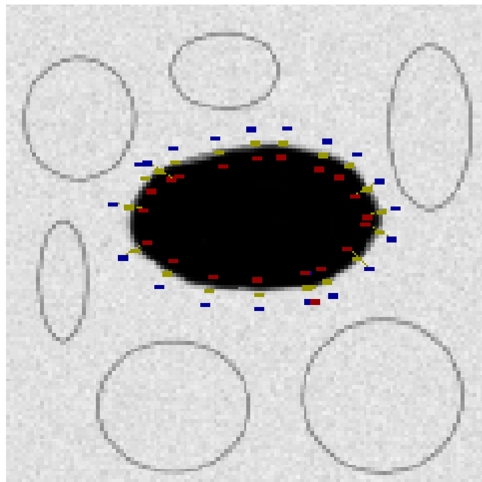
I have made two attempts to get the sample points in my application, and I will explain each of them in detail in the following words.

4.3.1 Using normal vector to determine positive and negative sample points

Using normal vector to determine the positive and negative sample points came out into my brain the first time when I start consider this problem. The main idea of getting positive and negative sample points is very obvious: for each B-Spline curve in the data slice, after I adjust the curve to make it fit the edges of the image, I:

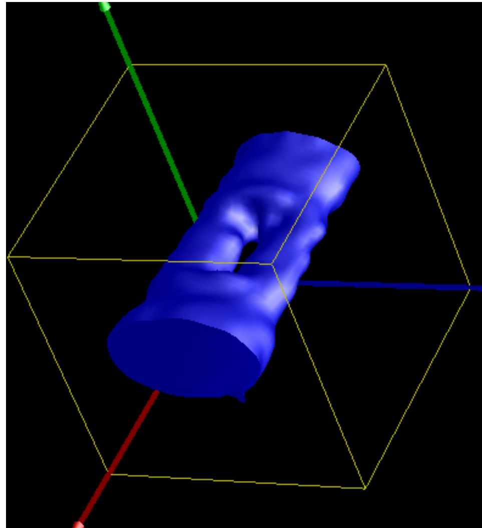
- take 20 sample points which has similar distance between every two of them by using the *t* parameter of the B-Spline (Problem 1 introduced in the section 3.2.3), and assign zero field value to all of them
- determine normal vector at every zero-valued sample point by working out the tangent vector at that point first
- take 2 points in each side of the zero-valued sample point along the normal vector with a constant distance
- separate into situations to determine which point should be assigned to positive field value and which negative

Like the picture in the following:



Picture 4.3.1.1: Sample points from attempt 1

In the above picture, the grey points are zero-valued sample points directly taken from the curve, red points are the positive sample points, and blue points are negative sample points. I assign the field values of all red points to 10, and blue points to -10.
So I use these sample points to create the final model by using the Radial Basic function, the picture of the final model is in the following:



Picture 4.3.1.2: the final model created from attempt one

There are several disadvantages of attempt one:

- The way to determine the positive and negative sample points are not correct all the time in my application. As I just use the normal vector and a constant distance value to determine the positive and negative points. These way works for some simple model like above, but it will get a big problem if I have very complex and messy picture. Then it will be very hard to determine which point inside the closed curve and which is outside. This is a common problem in the mathematic as well.
- Too many sample points will be created by this way. Because I will create 2 points for each zero-valued point, 20 zero-valued points will be taken from one curve, and there are at least 15 closed curves. Totally, there will be 3*20*15 = 900 sample points for every test case. 900 sample points are quite a lot for the LUSolver. Sometimes LUSolver will make some numerical error while working out the coefficients of the Radial Basic Functions (Because while the number of sample points getting bigger, some numerical error will occur when a bad processor was included in the running of a parallel application). Another problem caused by these 900 sample points is that it's quite time-consuming while do the calculation.

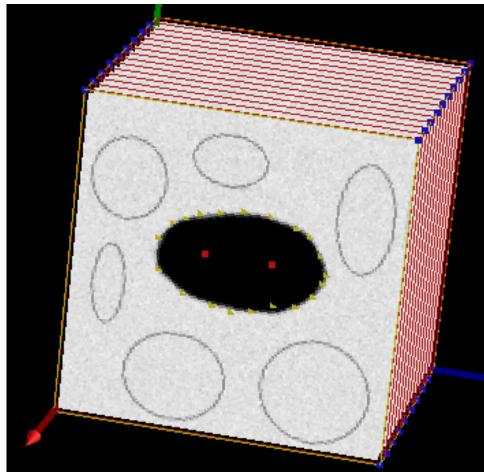
Because of the two disadvantages above, I have to find another way to efficiently and effectively work out the sample points for Radial Basic Function and display the final model. This make me think of the way I used in attempt two in following section.

4.3.2 Using initial skeleton of the soft object to determine the positive sample points

In my second attempts, I use the skeleton of the initial soft objects created in the environment to determine the positive sample points, the reason of doing this are:

- The soft object models approximately simulate the edges of the models formed by the data images, the skeletons of the soft object models are in the best positions for the positive points.
- Using skeletons to define the positive sample points can lead to much less sample points than previous attempt (In fact I don't need to specify a positive and a negative points for every zero-valued sample point, even for every slice).

But there is another problem arising: because I am using the 3D Marching Cube to display the final reconstructed model, positive points and zero-valued points are not enough to help Radial Basic Function precisely determined the field values of every points in the cube (the reason will be introduced in the section 4.3.3). Then I need to determine the negative sample points as well, and this leads to another difficulty that where and how many negative sample points I need to define. In my application, as the image data are along the z axis of the coordinate system, I define the points along the edges which are parallel to the z axis as the negative points. This obviously is not a sufficient way to determine the negative points, but it work for most of the case with only a few faults. The picture below shows the sample points selected by method 2:

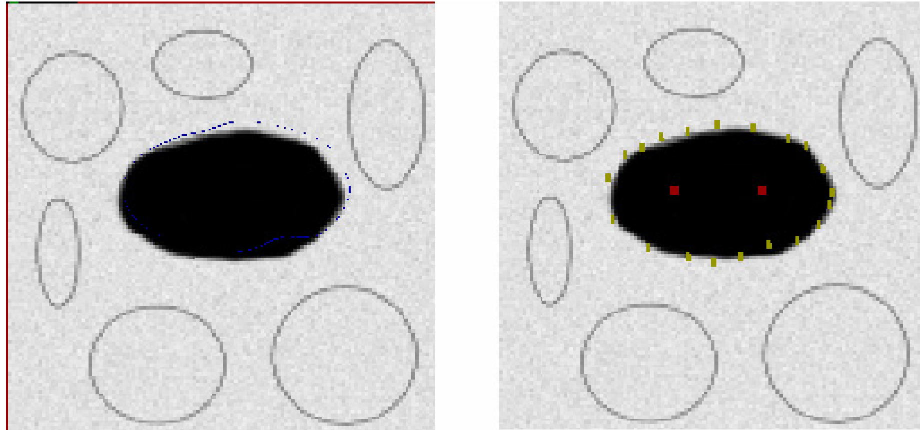


Picture 4.3.2.1: The sample points from method 2

In the above picture, we can find that at most 2 positive sample points in each slice and all negative sample points are in the edges paralleling the z axis. In the test case above, finally there are 20*15 + 10*12 = 420 sample points, this is less than half of the sample points from attempt one, it's much efficient and effective.

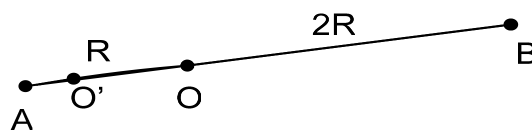
4.3.3 A common problem in both

There is a common problem appears in both of the attempt mentioned before, let have a look at the following pictures, they are the meshes mode and sample point's mode of the final model:



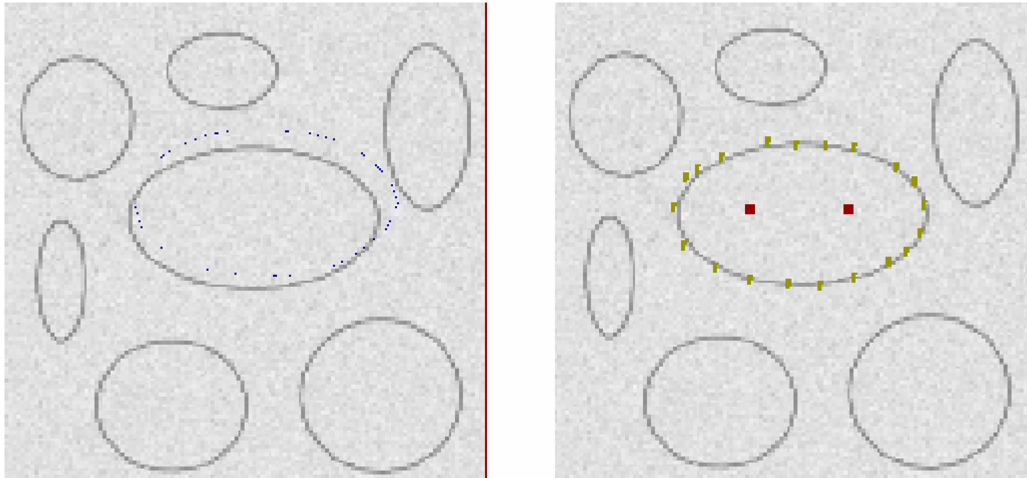
Picture 4.3.3.1: The mesh mode and sample point's mode of the final model from 3D Marching Cube with intensity 40

For the left picture above, the blue curve is the edges of the final model in current slice. The right picture is a picture of the sample points for the final model. As I am passing the sample points to the Radial Basic Function, and re-constructing the final model by 3D Marching Cube with 0 field values specified, the blue curve should exactly go though all the yellow points (zero-valued sample points) in the right picture, but actually they are not.
I have substituted the zero-valued sample points into the Radial basic function again to see whether the field value is zero (if not, then that means there is some problem in my code). The result of this testing is that all field value for zero-valued points are very close to zero, so this is not the cause of the problem.
Another cause of the problem I guess is that: as the way 3D Marching Cube to work out the zero-valued point from two other valued points at each side is using linear interpolation directly, however the way radial basic function use to determine the field values is not linear, it's about n^2*log(n). This could lead to the problem of not going though the zero-valued points.
For example, like the graph below:



Point O is the point with zero field value, and point A is a positive point, point B is a negative point, the distance between A and O is R, and the distance between B and O is 2R. If the field value in A is x, so that the field Value of point B should be -2^2log(2)*x. However, if I know the position and field values of the points A and B and want to work out the zero-valued point by linear interpolation, it will give me the point O' but not O. This is the reason why the meshes curve didn't go though the sample points. A way to prove my guess is right is by changing the intensity of the Marching Cube: If I decrease the resolution of the Marching cube, the gap between curve and sample points should be bigger. The following pictures show

the meshes curve and sample points of the final model after I decrease the resolution of the Marching Cube:



Picture 4.3.3.2: The mesh mode and sample point's mode of the final model from 3D Marching Cube with resolution of 20 by 20 squares for the image

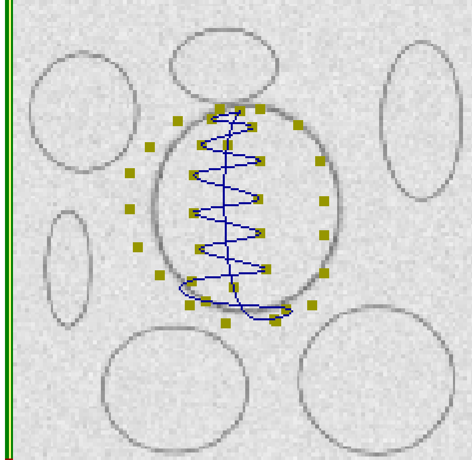
According to the pictures above, I can obviously find out that the curve of the mesh mode is far away from the sample points compared with the Pictures in 4.3.3.1. This means the Marching Cube must be one of the causes of the problem.

5. Sample test case for my application

Before I introduce the sample test case of my application, I will introduce another technical problem I solved in my application first.

5.1 Re-arrange the control points to form the closed B-Spline curve

In my application, after the users use soft object models to approximately simulate the shape from data image, my application will take the corresponding points and field values for each slice created by soft object, and it uses a 2D marching Cube to work out the corresponding interpolated points. So now I have a problem to re-arrange the interpolated points before I use them to create *BSpline* instance. The reasons why I need to re-arrange the interpolated points is that they are not in the correct order initially, if I don't re-arrange them, a very messy curves will be created. Like the picture in the following:



Picture 5.1.1: The curve created before rearrange

According to the above picture, we can find that if we don't correctly arrange the control points of the B-Spline curve, a not correct curve will be created, and it definitely can not be used for the future adjusting.
I re-arrange the control points by writing a algorithm which connecting every head and tail of the point pairs returned from the 2D marching cube. The detail of these lines of code are in the `applyMarchingSquare()` method, and all the methods related to it.

5.2 Test case: Constructing the Pelvis of the human body

Finally, in order to show the effective and efficiency of my application, I provide a test case together with my application, this is a test case of reconstructing human's pelvis by my toolkit. I take the sample MRI data set of Human's Pelvis directly from the Auckland University COMPSCI 716's assignment web page. This is a real MRI dataset of a human's pelvis, and this dataset consists of 56 image slices in grayscale color (value between 0 and 255), so I use these image slices as the basic slices users should follow while approximately constructing the model in the data set. A quick look at of these image slices is in the following:

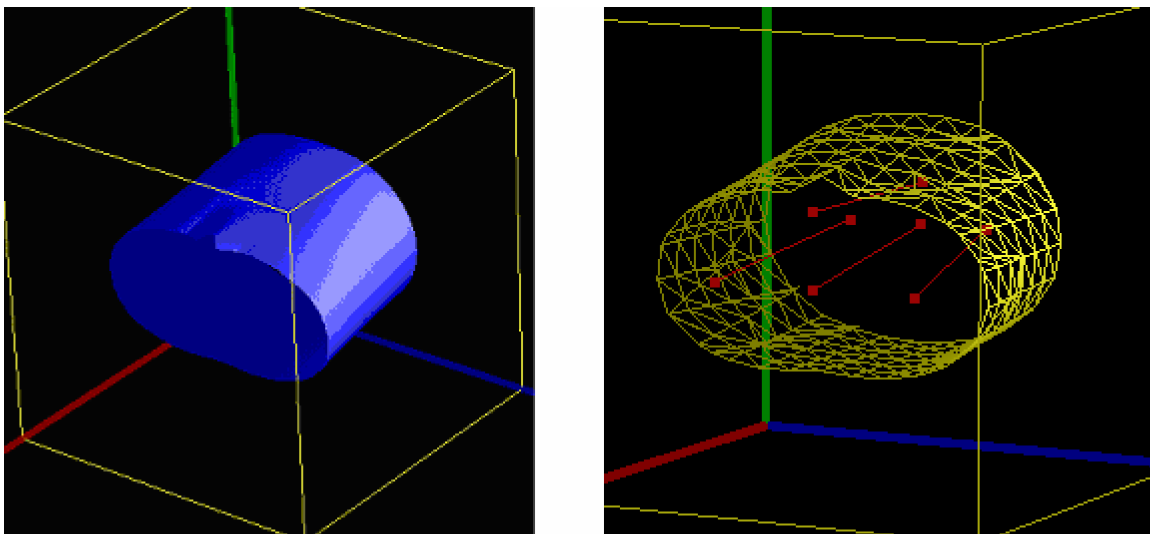
Picture 5.2.1: The MRI data slice of human's pelvis

Initially, I planned to make a simulated model of the bones in the pelvis (like the parts inside the red circles of the above image). But the intensity of the Marching Cube in my application is not high enough to precisely detect the edges of bone in this pelvis dataset. Then I changed my mind to construct the whole outside skins of the pelvis other than the bones.

For the Pelvis MRI data set I get from the internet, because the width, height and length of this data set are 128,128, and 56, it only takes about half of the working environment to display all MRI data slices (because the length is only 56, about half of 128). I have modified my application to limit the display points so that they can only be displayed in this limited area. In my application, the area user use to construct the pelvis is all the space from 0.25 to 0.75 in z axis.

The description about the steps I make my test case is in the following:

- I hard-coded 4 Soft Cone models into my application to approximately construct the outside skin of the pelvis (In normal situation, users are required to do these steps by mouse by themselves).



Picture 5.2.2: 4 Soft Cones which approximately simulate the pelvis

The positions and radius values of these 4 soft cone models are:

Soft Cone 1:

Position: `(0.33, 0.51, 0.25) (0.27, 0.44, 0.75)`

Radius: `1.5 1.3`

Soft Cone 2:

Position: `(0.65, 0.51, 0.25) (0.73, 0.44, 0.75)`

Radius: `1.5 1.3`

Soft Cone 3:

Position: `(0.5, 0.51, 0.25) (0.5, 0.44, 0.75)`

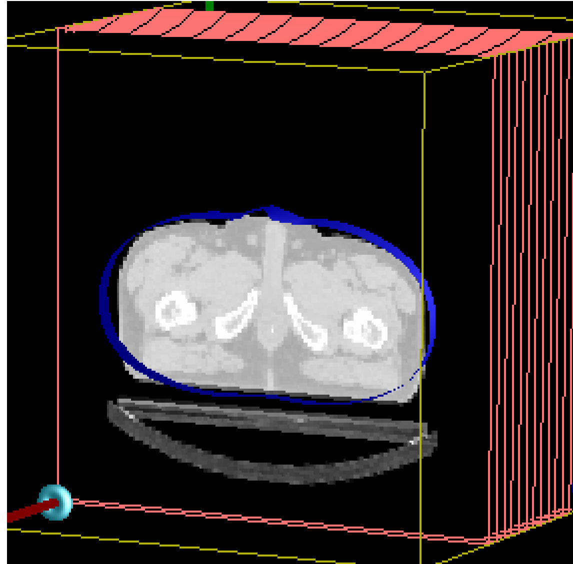
Radius: `1.5 1.3`

Soft Cone 4:

Position: `(0.5, 0.6, 0.25) (0.5, 0.6, 0.75)`

Radius: `0.3 0.3`

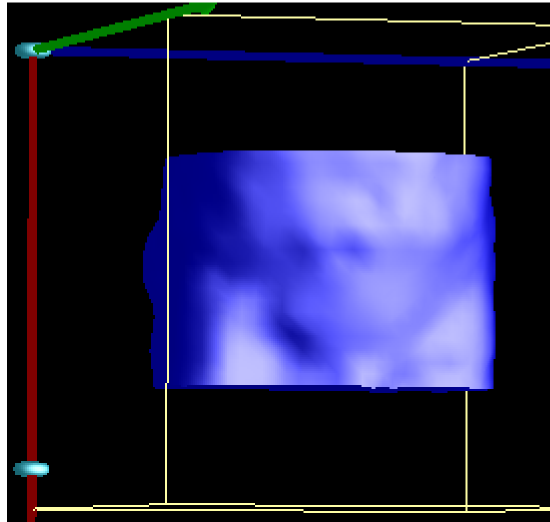
- With the Soft Cone models mentioned above, we get a approximate model of the human's pelvis. For the picture below, we can find that the contour of the model formed by 4 soft cones are still not exactly match the edges of each slices, it just approximately close to them.



Picture 5.2.3: Model formed by4 Soft Cones to approximately simulate the pelvis

- After the approximate model is created (according to previous two steps), I click the `GetSnake` Button in my application. This button actually starts all the processes of computing and displaying the final model described by the MRI dataset. They are listed in the following:
    1. Get the field values in the environment created by soft object
    2. Take the corresponding field values to form a 2D array for each data slice
    3. Pass fields values from previous step to a 2D Marching cube to work out the interpolated points of the current environment with the data slice
    4. Use the interpolated points to form a closed B-Spline curve by passing them into a new *BSpline* instance
    5. Call the `adjustBSpline()` method for every *BSpline* instance to adjust the initial B-Spline curve and make them match the edges in every image slice.
    6. Call the `getSamplePoints()` method in *BSpline* instance to get the zero-valued sample points from each adjust curves, also get the positive and negative sample points as well.
    7. Pass all the valued sample points into the *LUSolver* class, and call the `display()` method in the *LUSolver* class to get the new field values.
    8. Using 3D marching cube again to reconstruct the model by the new field values.

- finally, I got the human's Pelvis like following picture:



Picture 5.2.4: Final model of human's pelvis from my application

After this human's pelvis is displayed, users still can switch between this final mode and the initial soft cones mode by using the 'a' button in the keyboard.

5.3 Contained problem and further improvement of my application

Currently, the contained problem in my application is still the problem that the final curve of my model doesn't match the zero-valued sample points (I mentioned in section 4.3.3).
One improvement of my application can be made in the process in which I get the interpolated points for each slice in my application. Currently I get these points from the field values which are pre-calculated and stored in my application already. This is not precise enough for using 2D Marching Cube to get the interpolated points in each data slice. I can change this process by recalculating the field values at each slice singly, and then I can get a very precise field values for my 2D Marching Cube.

6. Conclusion and further research

In this project, I have developed a program which can help the medical researchers much easier to track the contour in each image slice. The reason why my software is convenient is that the researchers can use the soft object models to approximately construct the shape in each MRI data slice, and get the exact contour by letting computer to work it out automatically, this is much easier than asking researchers to use mouse to track the contour themselves by hand.
In my future research, I can improve my application by finding a better way to reconstruct the model from the adjust curves other than using the Radial Basic

Function. The reason is that although the Radial Basic Function provide a really smooth surface for the model, it is quite time-consuming to run. I am currently focusing on using free form deformation technique to create deformable meshes to match the adjust curves from the B-Spline snake. This is quite a new technique in this area, however, it's still far away from finish.

Reference

[1] M. Kass, A. Witkin, D. Terzopoulos, Snakes: active contour models, International Journal of Computer Vision 1 (4) (1988) 321-331.

[2] Burkhard Wunsche and Ewan Tempero, A Comparison and Evaluation of Interpolation Methods for Visualizing Discrete 2D Survey Data.

[3] Carr, J. C., Beatson, R. K.,Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C. & Evans, T. R. (2001), Reconstruction and representation of 3D objects with radial basic functions, in 'Proceedings of ACM SIGGRAPH 2001', Computer Graphics Proceedings, Annual Conference Series, pp. 67-76.

[4] William H.Press, Saul A. Teukolsky, William T. Vetterling, Brian P.Flannery, LU Decomposition and Its Application, Chapter 2.3, Numerical Recipes in C.

[5] Bo Li, Modelling heart pathology with Soft Object Modelling, COMPSCI 380 project, supervised by Burkhard Wuensche, Department of Computer Science, University Of Auckland, New Zealand, November 2003.