Implementation of a Feature-Preserving Volume-Filtering Algorithm

Ivan Viola, Matej Mlejnek
{viola, mlejnek}@cg.tuwien.ac.at

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

1 Abstract

In this paper the implementation of a new feature-preserving volume-filtering technique is presented. The method is based on the minimization of a three-component global error function penalizing the gradient and density deviations and the curvature of the unknown filtered function. This method performs filtering in the frequency domain. Therefore, an effective 3D Fourier transformation was necessary to be implemented. We describe briefly the basics of the Fourier transformation and its optimizations. The filtering method was implemented in Matlab, Java and C. For the sake of clarity, we explain the reason why we have implemented these three versions, as well as the problems that have arisen during the implementation. We will describe the functionality of the analyzing tool implemented in Java and the optimized algorithm in C. At the end we shortly sketch some other application possibilities.

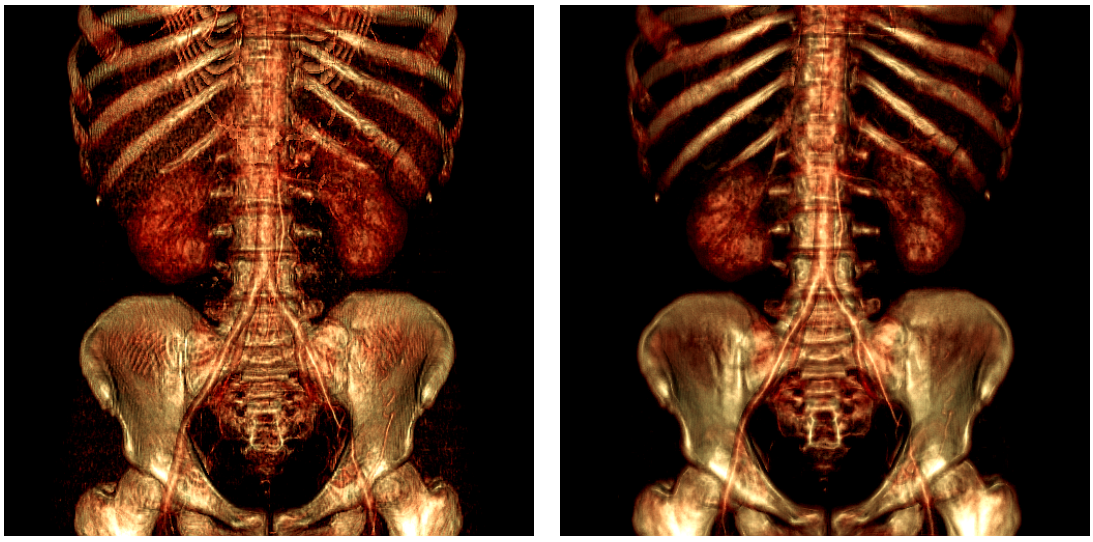
KEYWORDS: image processing, fast Fourier transformation, antialiasing, noise filtering, feature-preserving smoothing.

2 Introduction

In image processing it is a fundamental problem how to reconstruct features (like the original gradients) from sampled data. A typical example is the reconstruction of medical data obtained as CT or MRI scans. The data can be sampled at regular or irregular grids points. If we use direct volume rendering to visualize such a data set staircase artifacts can occur. In case of binary segmented data the staircase aliasing is stronger than in case of gray-scale data, where the gradients can be estimated more accurately. A binary volume can be obtained as a result of a segmentation. The segmented data cannot be rendered directly, because the result looks like built from Lego pieces. Therefore, various preprocessing techniques are used to eliminate this aliasing. The classical method is convolution-based filtering. Convolution-based methods have mostly local influence, because of the limited support of the kernel function. Although filtering with a wide kernel causes dependency on wider neighborhood, it is rather time-consuming and removes fine

details. There are several approaches for solving this problem. One research direction is interpolation oriented assuming, that accurate samples are available. The *sinc* and *cosc* functions are considered as ideal interpolation and derivative filters respectively. These derivative reconstruction techniques based on windowing are local methods as for practical reasons only a limited number of neighboring samples are taken into account. Another approach for derivative reconstruction is approximation oriented. Here it is assumed, that the sampled function is noisy, which is typical, when some real physical properties are measured. The basic idea is to estimate the inclination or the normal from a larger neighborhood. In order to reduce staircase aliasing, several methods were proposed for normal computation especially in binary volumes. Contextual shading techniques try to fit locally approximated plane or a biquadratic functions to the set of points that belong to the same iso-surface. These methods are time consuming and limited to a certain neighborhood. A rather new research direction is based on a gradient estimation using 3D distance maps.

The new method we have implemented, designed by Neumann et al. [4] is a general tool for filtering binary and gray-scale data sets. Figure 1 illustrates this filtering technique on a CT scan of a human body.



(a) (b)

Figure 1: Direct volume rendering of a human body using the original (a) and the filtered (S = 1, G = 1) data (b).

Similarly to the distance maps, generated from binary volumes, this filtering technique creates a smooth volume from gray-scale data. Unlike convolution based filtering, the smoothing effect is global due to a global curvature minimization. Feature preservation is the main characteristic of this novel filtering approach. By globally penalizing large gradient deviations, important features and fine details like edges or iso-surfaces are preserved. The method is based on a quadratic penalty function E. E is defined so that feature preservation and smoothing is

simultaneously possible. Therefore E consists of the following three components summed over every sample point i:

- difference squared between filtered value $\tilde{f}_i$ and original value f_i
- difference squared between the gradient of the filtered value $\tilde{f}_i$ and the original value f_i
- the squared curvature of the filtered function $\tilde{f}$

The 1D case of the penalty function E has the following structure:

$$E = \sum_i [(\tilde{f}_i - f_i)^2 + G \cdot [(\tilde{f}_{i+1} - \tilde{f}_{i-1})/2 - g_i]^2 + S \cdot (\tilde{f}_{i+1} + \tilde{f}_{i-1} - 2\tilde{f}_i)^2]. \quad (1)$$

The weights S and G determine the relative importance of feature preservation as opposed to smoothing. At the minimum location of the penalty function E the partial derivatives according to all the N unknown values $\tilde{f}_i$ have to be equal to zero, where $i = 0, 1, 2, ..., N-1$:

$$\partial E(\tilde{f}_0, \tilde{f}_1, \tilde{f}_2, ..., \tilde{f}_{N-1})/\partial \tilde{f}_i = 0. \quad (2)$$

As penalty function E is a quadratic function, the partial derivatives are linear functions of variables $\tilde{f}_i$. Therefore, having all of these partial derivatives evaluated, a large linear equation system is obtained with N unknown variables:

$$A \cdot \tilde{f} = m, \quad (3)$$

where A is a sparse coefficient matrix, and $\tilde{f}$ is the unknown vector containing the N samples of the filtered function. Vector m is derived from the original function samples f_i in the following way:

$$m_i = f_i - 2G \cdot (g_{i+1} - g_{i-1}). \quad (4)$$

Matrix A, derived from the partial derivatives, is a band matrix defined by a symmetric point spread vector p:

$$p = [p_1, p_2, p_3, p_4, p_5] = [S - G, -4S, 1 + 6S + 2G, -4S, S - G], \quad (5)$$

$$A = \begin{bmatrix} p_3 & p_4 & p_5 & 0 & . & 0 & 0 \\ p_2 & p_3 & p_4 & p_5 & . & 0 & 0 \\ p_1 & p_2 & p_3 & p_4 & . & 0 & 0 \\ 0 & p_1 & p_2 & p_3 & . & p_5 & 0 \\ . & . & . & . & . & p_4 & p_5 \\ 0 & 0 & 0 & p_1 & p_2 & p_3 & p_4 \\ 0 & 0 & 0 & 0 & p_1 & p_2 & p_3 \end{bmatrix}.$$

In fact, the multiplication of $\tilde{f}$ with coefficient matrix A is a convolution of the unknown vector $\tilde{f}$ with kernel p. Therefore, it is not necessary to use any computationally expensive linear algebra method, since a simple deconvolution leads to the solution. Such a deconvolution can be efficiently performed in frequency domain.

Using fast Fourier transformation, the filtering algorithm consists of the following steps:

1. estimation of gradients g_i using linear regression
2. non-linear operations on the gradient function g
3. calculation of function m using the modified g
4. $M = FFT(m)$ - Fourier transformation of function m
5. $P = FFT(p)$ - Fourier transformation of function p
6. $\tilde{F} = M/P$ - deconvolution in frequency domain
7. $\tilde{f} = INVFFT(\tilde{F})$ - inverse Fourier transformation of $\tilde{F}$

3 Fast Fourier transformation

Fourier transformation is a tool which generates the spectrum of a signal yielding a frequency-domain representation. Since this transformation is unambiguous the original signal can be reconstructed from its spectrum by an inverse transformation. The Fourier transform $F(u)$ of a 1D function $f(x)$ is defined as:

$$F(u) = \int_{-\infty}^{\infty} f(x) \cdot e^{2\pi i u x} dx, \tag{6}$$

where u is a value in the frequency domain. The inverse Fourier transformation for reconstructing $f(x)$ from $F(u)$ is defined as:

$$f(x) = \int_{-\infty}^{\infty} F(u) \cdot e^{-2\pi i x u} du \tag{7}$$

which is rather similar, except that the exponential term has the opposite sign. In the 3D case, the Fourier transform of a function $f(x, y, z)$ is defined as follows:

$$F(u, v, w) = \int\int\int f(x, y, z) \cdot e^{2\pi i (ux+vy+wz)} dx dy dz \tag{8}$$

The inverse transformation is analogous to the 1D case.

3.1 Discrete Fourier Transformation

But the images are digitized, therefore we need a discrete formulation of the Fourier transformation, which takes regularly spaced data values, and returns the coefficients of the discrete Fourier transformation as a set of equally spaced values in the frequency space. This is done by replacing the integral by a summation defining the discrete Fourier transformation (DFT). In 1D, it is convenient to assume, that the series outside the range 0, $N-1$ is extended N-periodic, thus $f(k) = f(k+N)$ for all k. The DFT of this series is denoted by $F(k)$ and represented by N samples. The DFT is defined as follows:

$$F(n) = \sum_{k=0}^{N-1} f(k) \cdot e^{2\pi ikn/N} \quad where\ \ n = 0..N-1 \tag{9}$$

while the inverse DFT is:

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} F(k) \cdot e^{-2\pi ikn/N} \quad where\ \ n = 0..N-1 \tag{10}$$

The 3D DFT is defined by $N_1 \times N_2 \times N_3$ samples in the frequency domain:

$$F(n_1, n_2, n_3) = \sum_{k_3=0}^{N_3-1} \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} f(k_1, k_2, k_3) \cdot e^{2\pi ik_3 n_3/N_3} \cdot e^{2\pi ik_2 n_2/N_2} \cdot e^{2\pi ik_1 n_1/N_1} \tag{11}$$

The inverse transformation is analogous to the 1D case:

$$f(n_1, n_2, n_3) = \frac{1}{N_1 N_2 N_3} \sum_{k_3=0}^{N_3-1} \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} F(k_1, k_2, k_3) \cdot e^{-2\pi ik_3 n_3/N_3} \cdot e^{-2\pi ik_2 n_2/N_2} \cdot e^{-2\pi ik_1 n_1/N_1} \tag{12}$$

The DFT can be applied to any complex series. The computational time is proportional to the square of the number of points in the series. Instead of the naiv implementation a much faster algorithm can be used, called FFT (Fast Fourier Transformation) [2]. DFT requires at least N^2 multiplications to generate all N of the coefficients $F(n)$. As it is explained in [5], the summation can be broken into two parts, one over the even-numbered elements ($k = 0, 2, 4, ...$) and the other over the odd-numbered elements ($k = 1, 3, 5, ..$). In turn, each one of these parts can be broken into its even-numbered and odd-numbered parts, and the process can be continued, with careful book-keeping, until the summation has become divided into 1-point Fourier transforms (which are identity transforms). This repeated dissection of the series into even- and odd-numbered parts can be implemented by reversing the bit-pattern of the addresses of the data elements. After this is done, the required N values of $F(n)$ can be generated by making a sequence of $2, 4, 8, ...$-point summations. Here the number of points in the series is assumed to be a power of 2. Therefore we arrange always our data sets in

the middle of 2^n zero pad. The complexity of the algorithm is $O(N \cdot log(N))$. For example, a transformation of 1024 points using the DFT takes 100 times longer than using the FFT. Note that in practice comparing speeds of various FFT routines is problematic. Many of the reported timings have more to do with specific coding methods and their relationship to the hardware and operating system. Usually the signal to be digitized is appropriately filtered before sampling to remove higher frequency components. If the sampling frequency is not high enough the high frequency components wrap around and appear in other locations in the discrete spectrum.

Therefore we have to rearrange our data set after every Fourier transformation. This can be done by dividing the volume data into eight parts (similar to the octree subdivision) numbering them and storing them in the reverse order. The 2D case is sketched on the figure 2.

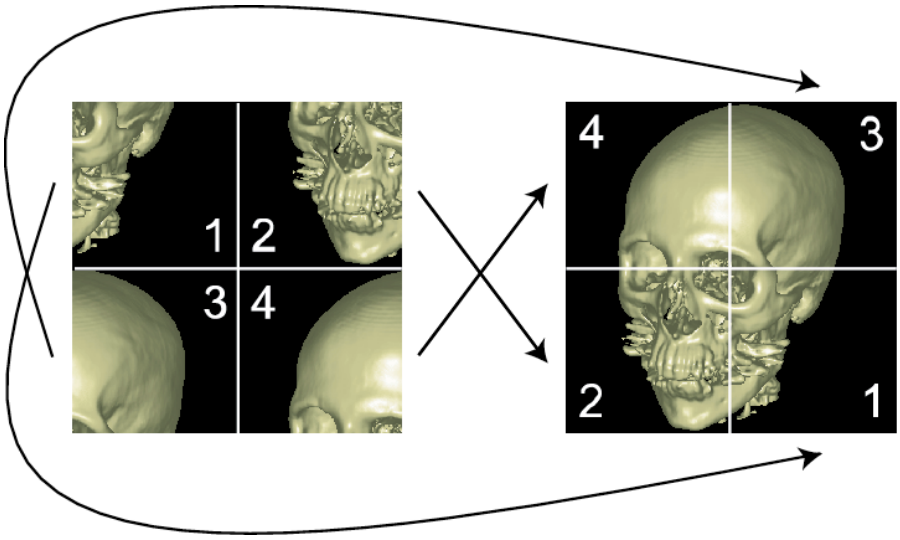


Figure 2: Wrap around effect: the high frequency components wrap around and appear in other locations of the discrete spectrum. This can be solved by rearranging the different parts.

The Fourier transformation is designed only for periodic signals. In image processing one might need to analyze non-periodic signals as well. Then the whole signal (e.g. row of pixels) is considered as only one period of the signal. This may cause that the values at the end of the period are influenced by values from the beginning. If we take a picture as an example, this means, there will be some artifacts at the border. One effective solution is the use of zero-padding.

4 Motivation of various implementations

Our implementation was influenced by a lot of aspects. Generally we wanted to implement the method in Java, because of its object oriented clarity and programming comfort. Later, when we realized, that we need not only the algorithm

implementation itself, but also a kind of analyzing tool, we appreciated this decision, because of the simple GUI design in Java. The first task was to implement a fast and efficient Fourier transformation. This is actually not a big problem, there are lot of sources. The bigger problem was, how to check if the Fourier transformation really works, as it should. To check the 3D version of the FFT is not that simple. Actually the only method, which gives quasi 100% certainty, is to compare it with already implemented Fourier transformation value by value. As a reference we have chosen Matlab implementation of FFT. Another problem of our Java implementation was, that it was more a tool for analysis of the smoothing technique, than a fast implementation. Generally all computationally expensive algorithms are slower if they are implemented in Java. That was the reason, why we have decided to make a fast and optimized C version to achieve the best performance.

5 Java implementation

The real object-oriented approach was one of the main reasons, why to code in Java. It was obvious from the first moment, that this will help to keep the code clear and secure. For instance, our implementation was even more comprehensible than the C source downloaded from the Internet [1], because of using another class ComplexNumber, where the basic operations of complex computing were implemented, instead of using two float arrays.

5.1 Data Structures

It is obvious that the main structures are object-instances of various classes. This classes have implemented some methods, where the program functionality is stored. Here is a short overview of our classes:

- MaskSmoothing (public) - The "main" class. Actually it calls the AppFrame class and starts the application.
- AppFrame - This class is a subclass of the already implemented java.awt.Frame class. Here the GUI is implemented, reading and writing the data set. Therefore, all the data sets are finally stored in this class. Also the dominant part of the data visualization is implemented here.
- CanvasSlice, CanvasPerspective - Classes derived from the java.awt.Canvas. These are also used for the data visualization. CanvasSlice simply flushes the input image to the canvas. CanvasPerspective shows a box and three rectangles that represent the top, front and side slice position within the volume.
- AppFrame_AboutBox, AppFrame_SettingsBox, AppFrame_StatisticsBox - Classes of the about, settings and statistics dialog.

- ComplexNumber - One of the things, that are in Java not yet implemented, is the complex computing. Therefore we had to implement it ourselves, to perform mathematical operations in the frequency domain. There are not all of mathematical operations implemented in this class, we have implemented just those methods we needed for the calculation.

- FilterKernel - The class creates a filter according to the settings dialog.

- FilterMaskSmoothing - This is the core of the algorithm calculation. There are FFT, convolution, deconvolution and gradient estimation implemented in this class. For data representation we use one, and three dimensional arrays. For specific operations the 1D representation is preferred, where $f[x][y][z]$ is equal to $g[(z \cdot maxY + y) \cdot maxX + x]$. Another problem is, that Java assumes Little-Endian byte order, but our data set was stored in Big-Endian byte order. Unfortunately it is not possible to specify unsigned property of the basic types. Therefore, a short variable is read from the file as follows:

  ```
  x=0x7fff&(input.readUnsignedByte()|input.readUnsignedByte()<<8);
  ```

  where the input is an object of class DataInputStream, which is used for reading. As standard input type we took Big-Endian short (16 bit integral type). The reason is, the output of a CT contains also short integers although only 12 bits are used.

5.2 Data Visualization

The tool used for the visualisation is slice-based. It means, it is possible to view the separate slices in all three orthogonal directions. To use the maximal intensity range, the minimum and maximum density values of all the data sets (original, convoluted and deconvoluted) have to be found. Then a gray-scale ramp is divided to the whole density range. We tried different color/intensity assignments, for example two thresholds (for totally white and black) and between linear ramp or a kind of colorific representation (small density bluish, big density reddish), but the basic method was the clearest. For performance reasons, the nearest neighbor interpolation is used for finding the appropriate density value. If some slice has to be viewed, the values are first interpolated according to the actual canvas size and then stored in a 1D array which is displayed by the canvas.

5.3 Functionality

The working area of the application is divided into four parts, three othogonal views and one perspective view, where the relative position of orthogonal views is shown (see the figure 3).

But this can be changed in the menu "View". In order to see some details only in one orthogonal view, it is possible to switch to this type of view, and it is extended to the whole working area. The orthogonal views consist of two drawing areas to be able to see simultaneously two types of data sets. To move

within the slides in one particular direction, the user should move the slider of the scrollbar. This is initialized to a maximal value. The slider size changes after the filtering computation. This effect and also the label in the status bar at the bottom indicates that the computation is finished. The scrollbar range is set to the number of slices in the corresponding direction. This means, one click at the scrollbar button causes that the neighboring slice of the current one will be shown. The change of the viewed slice is immediately visible also in the perspective view and in the status bar, where the number of visible slices is shown. This features make the interaction with the tool user friendly, but this is not the most important property. What makes the tool really powerful is the possibility of choosing and setting various filters for convolution and deconvolution.

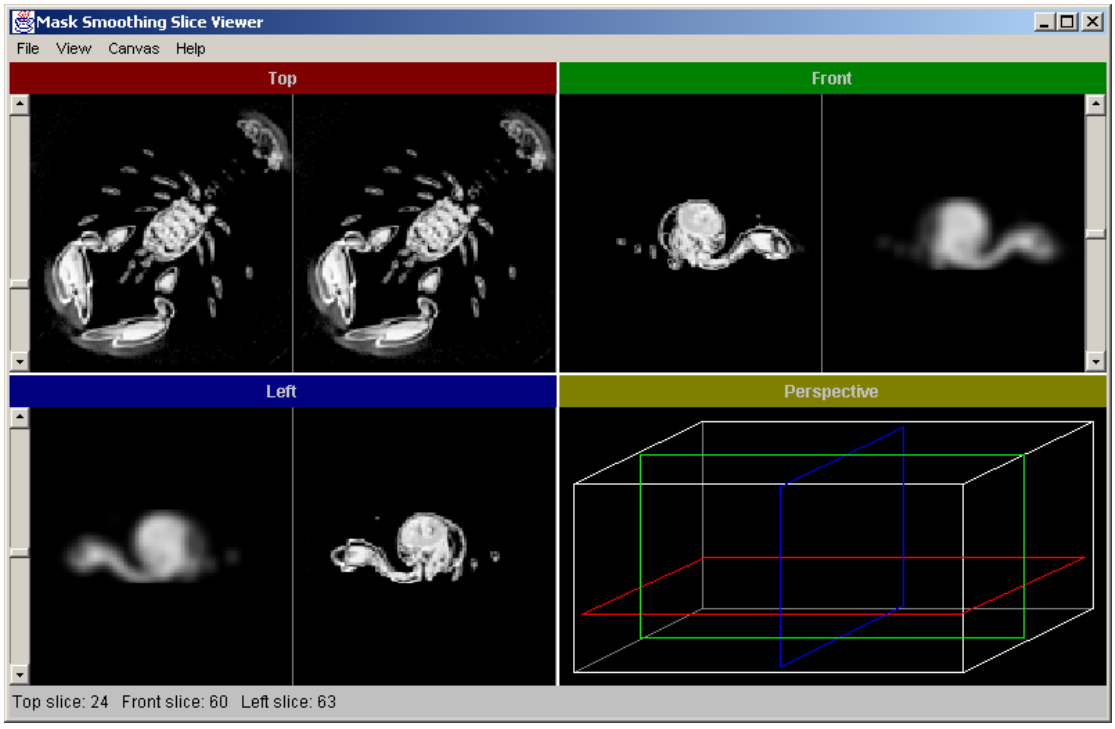


Figure 3: Filtering a CT scan of a lobster. Top view shows the original and the filtered data. Front view and left view show the corresponding convoluted and deconvoluted slices respectively.

This is very useful for the algorithm analysis and comparison with already existing methods used for smoothing. There are four categories of filters, where the sum of the weights is 1:

- *Basic Cubic Filter* - This is the simplest type, it calculates the average value of a cubic voxel neighborhood
- *Gaussian Cubic Filter* - Gaussian-like filter, where the weigth of a neighboring voxel is the reciprocal of its Manhattan distance from the current voxel.
- *Gaussian Spherical Filter* - Similar to the previous type, but instead of Manhattan distance an Euclidean distance is calculated.

- *Mask Smoothing Filter* - Filter kernel of the algorithm we have implemented. The dimensions are $5 \times 5 \times 5$. The first three filter types can be modified by changing the filter dimensions and the Mask Smoothing Filter is parametrized by S (smoothing) and G (feature preservation) parameters. This settings can be changed in the Settings dialog in the File menu. There are also some gradient settings in order to improve the results of the feature-preserving algorithm:
  - *Threshold* - below this value the gradients are considered to be zero.
  - *Multiplier* - scalar value used for emphasizing the gradient values.
  - *Type* - central differences or 4D linear regression [3].

Actually only these parameters have to be set before the computation. If the filter is already stored in a file, it is possible to reload it. This can be done by selecting the Open Filter menu item from the File menu. Also the data file has to be loaded analogous with Open Filter but Open Datafile menu item has to be chosen. The computation starts by clicking on the Run menu item in the File menu. When the computation is finished, it is possible to move slice-by-slice within the data set. As already mentioned, it is possible to view two data sets in one direction simultaneously (always the same slice). If one of the first three filter kernels was chosen for the computation, there is a possibility to change the data sets to be shown. For example in the Top view the original and the deconvolved data sets are shown, in the Front view convolved and the deconvolved data sets and original and the convolved data sets. All this can be customized in the Canvas menu. For some error or computational time information, the Statistics menu item from File menu item has to be chosen. It is also possible to save the deconvolved data set into a file. The last menu is Help, where the Online Help and About menu item are available.

As it can be seen, this tool gives the user a lot of possibilities for analysis, also looking for the optimal S, G and gradient parameters. In the "slice show" using one magnified view the user can quickly recognize if some structures are lost (because of smoothing) or not.

6 Matlab implementation

Matlab is a professional tool used for simulations and advanced computing. The big advantage is the simplicity of computational programming and fast visualization possibility using various graph representations.

In Matlab, we have modelled a couple of 2D cuts of basic objects (circle, square, ramp). Then we computed the fast Fourier transform of these samples and compared them to P.Bourke's Fourier transformation [1] implemented in our algorithm. We measured an average relative error lower than 0.1%.

7 C implementation

The Java version was designed for analysis of the smoothing technique. The Matlab version was implemented to check the accuracy of our results. We also made a fast and optimized C version to achieve the best computational time (the concept we took over from the Java Mask Smoothing tool). The C version is a simple non-interactive console application, so it is designed for batch mode processing, for creating pictures for animations, with varying parameters. For the sake of efficiency, it is a pure C language implementation. The data structures used in this version are rather basic types as float and short arrays. The volume data as well as the filter data are stored in 3D short arrays and processed in 3D arrays of complex numbers. A ComplexNr is a structure with additional functions such as divideComlexNr and multiplyComplexNr, needed to perform FFT fast and clearly. It consists of two floats: the real and imaginary parts of the complex number. This application can be executed from a window command shell as follows:
msc.exe filename s g gradient_type threshold
where the parameters are:

- *filename* - volume data set which has to be filtered
- *s* - smoothing parameter of the Mask Smoothing filter
- *g* - feature preservation parameter of the Mask Smoothing filter
- *gradient_type*:
  - *0* for central differences
  - *1* for linear regression 3x3
  - *2* for linear regression 5x5.
- *threshold* - if gradient size is smaller than this value, then it is considered to be zero.

Comparing to the Java version the C version is approximately 4-6 times faster. The entire filtering process took 8 minutes for a 202x152x255 resolution volume (see Figure 1) on an 800Mhz Pentium III PC with 512 MB RAM.

8 Summary

In this paper an implementation of a feature-preserving volume filtering method has been presented. This method is based on a minimization of a three-component penalty function, so that approximation of the original values, feature preservation and curvature minimization can be controlled efficiently. The scalability is ensured by the weighting parameters of the three-component penalty function. Due to the applied FFT method filtering is performed efficiently. We have implemented a tool, written in Java, to analyze this method by setting various weighting parameters and comparing it to other smoothing methods. Also a C version has been developed to achieve the best computational time.

9 Future work

A further application is sharp image zooming using gradient-based interpolation. From the original $N \times N$ gray-scale image we want to generate an $(s \cdot N) \times (s \cdot N)$ zoomed image. Using traditional resampling methods, the following problems arise:

- linear interpolation would result in blurred edges
- nearest neighbor interpolation would cause staircase artifacts

Therefore the goal is to obtain smooth and sharp edges inside a cell, where the interpolation is performed. The basic idea behind the gradient based interpolation is, that the original image is interpolated in each cell on an $s \times s$ subgrid. At the corner pixels the gradients are estimated from the original pixels (central differences or linear regression). The gradients at the subgrid points are calculated from the gradients of the four corner pixels using bilinear interpolation. The FFT method is performed on the entire $(s \cdot N) \times (s \cdot N)$ image (the gradients are known for each pixel and the intermediate pixel values can be defined as a constant (e.g. the average of the four corner voxels). In the FFT method relatively high S and G parameters are used. The penalty function is defined like in the 2D dedithering case [4].

References

[1] P. Bourke. DFT, Discrete Fourier Transformation. `http://astronomy.swin.edu.au/pbourke/analysis/dft/`, 1993.

[2] J. W. Cooley, J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation, 19, 90* pages 297–301, 1965.

[3] L. Neumann, B. Csébfalvi, A. König, E. Gröller. Gradient Estimation in Volume Data using 4D Linear Regression. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2000)*, pages 351–357, 2000.

[4] L. Neumann. Feature Preserving Smoothing. *TR-186-2-00-24 Institute of Computer Graphics, Vienna University of Technology*, 2000.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. Numerical Recipes in C. *Cambridge University Press*, 1992.